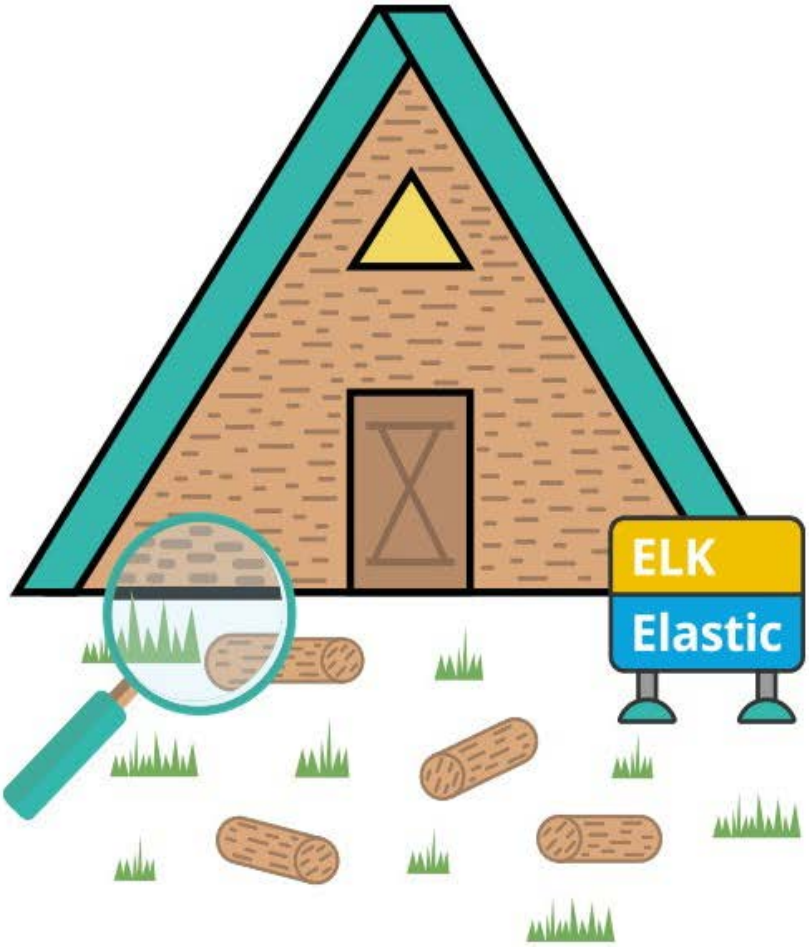
# Elasticsearch

## Consumindo dados real-time com ELK




Casa do Código

ALEXANDRE LOURENÇO

© Casa do Código

Todos os direitos reservados e protegidos pela Lei nº9.610, de 10/02/1998.

Nenhuma parte deste livro poderá ser reproduzida, nem transmitida, sem autorização prévia por escrito da editora, sejam quais forem os meios: fotográficos, eletrônicos, mecânicos, gravação ou quaisquer outros.

*Edição*

Adriano Almeida
Vivian Matsui

*Revisão*

Bianca Hubert
Vivian Matsui

[2016]

Casa do Código

Livros para o programador

Rua Vergueiro, 3185 - 8º andar

04101-300 – Vila Mariana – São Paulo – SP – Brasil

www.casadocodigo.com.br

# AGRADECIMENTOS

Agradeço a Deus pela minha vida e por tudo que possuo, e a meus pais, Maria Odete Santos Lourenço e Eleutério da Silva Lourenço, que me suportaram e proporcionaram a base que me permitiu chegar onde estou hoje. Sem vocês, eu não seria nada! Agradeço também a minha irmã Lucebiane Santos Lourenço e minha namorada Ana Carolina Fernandes do Sim, cujo apoio e carinho são fundamentais na minha vida.

Agradeço aos leitores do meu blog (http://alexandreesl.com), por me acompanharem nas minhas "aventuras" no mundo da Tecnologia. Ter um blog é uma experiência muito gratificante, que pretendo continuar por muitos anos.

Agradeço a meus colegas e ex-colegas de trabalho, Victor Jabur, Élio Capelati, Cláudio Dias Marins e Cristiano Sanchez, cujo apoio foi fundamental para este e muitos outros projetos na minha carreira!

Agradeço ao pessoal da Casa do Código, em especial ao Paulo Silveira, a Vivian Matsui e Adriano Almeida, cujo suporte me ajudou a escrever este livro. Desejo-lhes muita sorte e felicidade!

Por fim, agradeço a você, leitor, por depositar a sua confiança em mim ao adquirir este livro. Espero que eu possa cumprir as suas expectativas e fornecer uma boa fonte de informações sobre o indexador Elasticsearch e a stack ELK.

# SOBRE O AUTOR

Alexandre Eleutério Santos Lourenço é Arquiteto de Software, com bacharel em Ciência da Computação pela Pontifícia Universidade Católica de São Paulo (PUC-SP). Possui grande experiência na linguagem Java, com a qual trabalha desde 2003.

Como arquiteto, tem contato com diversas tecnologias, frameworks e linguagens, como Java, Python, C#, C/C++, Angularjs, HTML5, Apache Cordova, Spring, EJB, Hadoop, Spark e, é claro, ELK.

Apaixonado por tecnologia, filmes, seriados e games, possui o blog Tecnologia Explicada (*Technology Explained*), onde publica artigos sobre os mais diversos assuntos do gênero. Também possui alguns artigos publicados no site americano Developer Zone (https://dzone.com).

# PREFÁCIO

## A explosão dos dados

Vivemos em um mundo dominado por dados. Nunca foram produzidos tantos dados, de maneira tão rápida. Além disso, nunca se produziram tantos dados de maneira não estruturada, ou seja, que não seguem um modelo de estruturação formalizado, por meio do uso dos conhecidos `schemas`, como um XSD ou uma tabela em um banco de dados tradicional.

Para termos uma ideia do tamanho da massa de dados que temos em mãos, basta vermos os números de empresas como Twitter, Facebook e Netflix. Tais empresas processam milhões de dados por dia - em 2011, por exemplo, o Twitter publicou em seu blog que seus usuários alcançaram a marca de 200 milhões de tweets por dia! - e precisam de novos modelos para processar essas verdadeiras montanhas de dados, aproveitando-se de conceitos de computação distribuída.

Seguindo esses conceitos, diversas tecnologias foram criadas, como hadoop, spark, splunk, bancos NOSQL etc. Neste livro, abordaremos uma dessas novas ferramentas que permitem o processamento (consulta) de conjuntos massivos de dados textuais em tempo real, o Elasticsearch.

## Para quem se destina este livro?

Este livro se destina a desenvolvedores que desejam ampliar seus conhecimentos em Elasticsearch e seu ferramental relacionado, cujos casos de uso discutiremos no decorrer do livro. Para melhor aproveitar o livro, o leitor deve possuir algum conhecimento em REST e JSON, visto que as principais interfaces que temos

disponíveis para interagir com um cluster Elasticsearch se utilizam desses padrões de comunicação e formato de mensagens. Conhecimento básico da linguagem Java também pode auxiliar no entendimento.

## Como devo estudar?

No decorrer do livro, em alguns capítulos teremos *hands-on* e outros tipos de atividades práticas. Todo o código-fonte dessas atividades se encontra em meu repositório. Se o leitor desejar obter uma referência rápida:

https://github.com/alexandreesl/livro-elasticsearch.git

Convido o leitor a entrar também no Fórum da Casa do Código:

http://forum.casadocodigo.com.br

# Sumário

**1 Introdução** **1**

1.1 Conhecendo o Elasticsearch 1

1.2 Instalação 9

**2 Dissecando a ELK – Logstash** **13**

2.1 Criando pipelines de dados 13

2.2 Construindo nossa API de pedidos 14

2.3 Começando com o Logstash 19

2.4 Parseando as informações de log 26

2.5 Conceitos e outros plugins 37

2.6 Filtros condicionais 40

2.7 Conclusão 40

**3 Dissecando a ELK - Elasticsearch** **42**

3.1 Montando um cluster de buscas full text 42

3.2 Integrando as ferramentas 43

3.3 Entendendo a estrutura interna do Elasticsearch 48

3.4 Ações do Elasticsearch 53

3.5 Preparando a massa de testes com o Apache JMeter 54

3.6 Analisadores e scores de documentos 61

3.7 Consultas básicas do Elasticsearch 67

3.8 Plugins 75
3.9 Conclusão 77

**4 Dissecando a ELK – Kibana 78**
4.1 Desenvolvendo ricas interfaces para os nossos dados de log 78
4.2 Conhecendo o Kibana 78
4.3 Instalação do Kibana 79
4.4 Configurando o Kibana 80
4.5 Executando o Kibana pela primeira vez 83
4.6 Aplicações do Kibana 87
4.7 Conclusão 102

**5 Elasticsearch avançado 103**
5.1 Manutenção de índices 103
5.2 Manutenção de documentos 110
5.3 Montando os exercícios práticos 113
5.4 Realizando consultas parent-child 121
5.5 Aprofundando em analisadores textuais 125
5.6 Templates dinâmicos 130
5.7 Outros modos de consulta do Elasticsearch 132
5.8 Filtros e cacheamento de queries 138
5.9 Conclusão 140

**6 Administrando um cluster Elasticsearch 142**
6.1 Montando o cluster 142
6.2 Descoberta de nós (discovery) 143
6.3 Configurando o cluster: configurações no Logstash 144
6.4 Configurando o cluster: configurações no Kibana 145
6.5 O arquivo de configuração principal do Elasticsearch 145
6.6 Resolvendo o split-brain de um cluster Elasticsearch 148
6.7 Tunning 149

6.8 Backup & restore 151
6.9 Monitoração da saúde do cluster com o Watcher 155
6.10 Expurga com o Curator 158
6.11 Segurança com o Shield 160
6.12 Conclusão 162

**7 Considerações finais 163**
7.1 Cases de mercado 163
7.2 E agora, o que estudar? 166
7.3 Conclusão 167

Capítulo 1

# INTRODUÇÃO

## 1.1 CONHECENDO O ELASTICSEARCH

Figura 1.1: Logo do Elasticsearch

O Elasticsearch foi criado por Shay Banon em 2010. Baseado no Apache Lucene, um servidor de busca e indexação textual, o objetivo do Elasticsearch é fornecer um método de se catalogar e efetuar buscas em grandes massas de informação por meio de interfaces REST que recebem/provêm informações em formato JSON.

Para entendermos as vantagens de se utilizar um indexador para nossas informações, vamos começar com um exemplo bastante simples. Imagine que temos uma API REST que implementa um CRUD (*Create, Read, Update e Delete*) de clientes. Usaremos o Spring Boot para subir nossa API. Com o intuito de não tirar o nosso foco do assunto principal, vamos *mockar* os dados em vez de

utilizar um banco de dados, por questão de simplicidade.

Para começar, vamos criar nossa classe de domínio:

```
public class Cliente {

    private long id;

    private String nome;

    private String email;

    //getters e setters omitidos

  }
```

A seguir, criamos as classes `Application` e `ApplicationConfig`, responsáveis por configurar e inicializar o REST. Não se preocupe se você não conhecer o Spring Boot, você pode encontrar a API pronta dentro do meu repositório (https://github.com/alexandreesl/livro-Elasticsearch.git), na pasta `Capitulo 1`.

Veja a `Application.java`:

```
@SpringBootApplication
public class Application {
    public static void main(String[] args) {
        SpringApplication.run(Application.class, args);
    }
}
```

Veja a `ApplicationConfig.java`:

```
public class ApplicationConfig {

    @Named
    static class JerseyConfig extends ResourceConfig {
        public JerseyConfig() {
            this.packages("br.com.alexandreesl.handson.rest");
        }
    }

}
```

Por fim, criamos a classe `ClienteRestService` , que executa o CRUD de cadastro de clientes. Vamos começar pelo método de listagem de todos os clientes, bem como a criação da lista de clientes mockados:

```
@Named
@Path("/")
public class ClienteRestService {

    private static final Logger logger = LogManager.getLogger(Clie
nteRestService.class.getName());

    private static Map<Long, Cliente> clientes = new HashMap<Long,
 Cliente>();

    private static long contadorErroCaotico;

    static {

        Cliente cliente1 = new Cliente();
        cliente1.setId(1);
        cliente1.setNome("Cliente 1");
        cliente1.setEmail("customer1@gmail.com");

        Cliente cliente2 = new Cliente();
        cliente2.setId(2);
        cliente2.setNome("Cliente 2");
        cliente2.setEmail("customer2@gmail.com");

        Cliente cliente3 = new Cliente();
        cliente3.setId(3);
        cliente3.setNome("Cliente 3");
        cliente3.setEmail("customer3@gmail.com");

        Cliente cliente4 = new Cliente();
        cliente4.setId(4);
        cliente4.setNome("Cliente 4");
        cliente4.setEmail("customer4@gmail.com");

        Cliente cliente5 = new Cliente();
        cliente5.setId(5);
        cliente5.setNome("Cliente 5");
        cliente5.setEmail("customer5@gmail.com");

        clientes.put(cliente1.getId(), cliente1);
        clientes.put(cliente2.getId(), cliente2);
```

```
        clientes.put(cliente3.getId(), cliente3);
        clientes.put(cliente4.getId(), cliente4);
        clientes.put(cliente5.getId(), cliente5);

    }

    @GET
    @Produces(MediaType.APPLICATION_JSON)
    public Collection<Cliente> getClientes() {

            logger.info("Foram buscados " + clientes.values().size
() + " clientes");

        return clientes.values();
    }

    @GET
    @Path("cliente")
    @Produces(MediaType.APPLICATION_JSON)
    public Cliente getCliente(@QueryParam("id") long id) {

        Cliente cli = null;

        for (Cliente c : clientes.values()) {

            if (c.getId() == id)
                cli = c;

        }

        logger.info("foi buscado o cliente " + cli.getNome());

        return cli;
    }

    //restante da classe omitida
```

A seguir, criamos os métodos para criação e alteração de clientes:

```
    @POST
    @Consumes(MediaType.APPLICATION_JSON)
    public void addCliente(Cliente cliente) {

        logger.warn("O cliente " + cliente.getId() + " foi inserid
o!");
```

```
        clientes.put(cliente.getId(), cliente);

    }

    @PUT
    @Consumes(MediaType.APPLICATION_JSON)
    public void mergeCliente(Cliente cliente) {

        contadorErroCaotico++;

        if ((contadorErroCaotico) % 7 == 0) {
            throw new RuntimeException("Ocorreu um erro caótico!")
;
        }

        logger.info("O cliente " + cliente.getId() + " foi alterad
o!");

        Cliente temp = clientes.get(cliente.getId());

        temp.setNome(cliente.getNome());
        temp.setEmail(cliente.getEmail());

    }
```

E por fim, o método para exclusão de clientes:

```
    @DELETE
    public void deleteCliente(@QueryParam("id") long id) {

        logger.info("O cliente " + id + " foi excluido!");

        clientes.remove(id);
    }
```

Para testarmos a API, sugiro que você utilize o Postman (https://www.getpostman.com). Com uma interface gráfica simples, é uma ótima forma de se testar APIs REST.

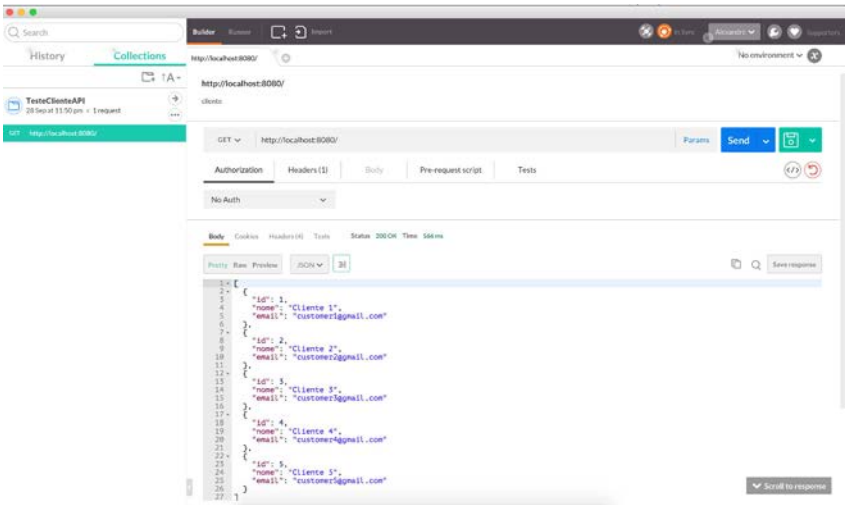



Figura 1.2: Postman em ação

Outra forma de testar as chamadas é a partir do comando linux/unix `curl`. Para simplificar as instruções, vamos usar o `curl` em nosso livro, mas encorajo você a experimentar o Postman, é muito bom!

Para executar a API, basta executarmos a classe `Application` da mesma forma que executamos uma aplicação Java comum, seja na nossa IDE favorita, como o Eclipse, ou mesmo pela linha de comando, com o bom e velho comando `java -jar <compilado do projeto>.jar`.

Vamos começar fazendo uma chamada simples. Abra um terminal e digite no seu prompt linux/unix:

```
curl http://localhost:8080/
```

O comando retornará uma estrutura JSON como a seguinte:

```
[{"id":1,"nome":"Cliente 1","email":"customer1@gmail.com"},{"id":2,
"nome":"Cliente 2","email":"customer2@gmail.com"},{"id":3,"nome":"
Cliente 3","email":"customer3@gmail.com"},{"id":4,"nome":"Cliente
```

```
4","email":"customer4@gmail.com"},{"id":5,"nome":"Cliente 5","emai
l":"customer5@gmail.com"}]
```

O leitor poderá notar que se trata da listagem mock que criamos no início da classe. Além disso, se notarmos no console do Spring Boot, vamos ver que uma linha de log foi criada:

```
22:05:04.055 [http-nio-8080-exec-1] INFO  br.com.alexandreesl.hand
son.rest.ClienteRestService - Foram buscados 5 clientes
```

Se você estiver seguindo o meu projeto de exemplo no Git, verá que também foi criado um arquivo `output.log` na raiz do projeto Eclipse, gerando a mesma linha de log mostrada anteriormente.

Antes de prosseguirmos com o estudo de caso, vamos a mais um exemplo. Vamos executar agora o seguinte comando no terminal:

```
curl http://localhost:8080/  -H "Content-Type: application/json" -
X POST -d '{"id":6,"nome":"Cliente 6","email":"customer6@gmail.com
"}'
```

Novamente, análogo ao exemplo anterior, veremos que foi criada uma linha tanto no console quanto no arquivo, como a seguinte:

```
22:22:43.902 [http-nio-8080-exec-1] WARN  br.com.alexandreesl.hand
son.rest.ClienteRestService - O cliente 6 foi inserido!
```

Tudo está ótimo para nós! Nossa API está operante, recebendo consumidores e efetuando nossos cadastros, além de gerar um log de execução que podemos utilizar para analisar as operações realizadas pela API. Você pode testar os outros métodos analogamente com comandos `curl` como os anteriores, trocando apenas detalhes como o `http method` ou os dados da request. É possível notar que também temos um gerador de erros caótico, que usaremos no futuro para simular erros na nossa API.

Um belo dia, porém, nosso chefe aparece com a seguinte demanda: *"Precisamos de uma dashboard para a diretoria que exiba,*

*em tempo real, indicadores de consumidores com mais alterações de cadastro e quantidade de novos cadastrados por dia, além de um gráfico de crescimento de erros da interface, para que possamos colocar em um painel de monitoração no nosso service desk".*

Em um primeiro momento, podemos pensar: fácil, basta fazer algumas queries no meu banco de dados e montar a dashboard em cima do meu banco de dados. Tal alternativa, porém, pode apresentar alguns problemas, como por exemplo, impacto na performance da base. No caso do gráfico de erros, a questão torna-se ainda desprovida de alternativas, visto que a única fonte de informações de erro que temos são nossos arquivos de log.

Existem no mercado diversas ótimas soluções que permitem resolver esse tipo de demanda, como o Nagios, por exemplo. Porém, esse tipo de solução costuma ser mais cara, demandando recursos de infraestrutura e operação que podem não se justificar para demandas de monitoração simples como a anterior.

É claro que esses não são os únicos cenários de sua utilização, e veremos no decorrer do livro outros exemplos. Entretanto, com este pequeno exemplo, o leitor já começa a ter uma ideia do que se pode fazer com o indexador, ainda mais se escorado pelo restante da stack ELK.

Uma regra de ouro que o leitor já pode ter em mente seria: *"Em cenários em que existe uma grande massa de dados textuais que necessita que sejam analisados padrões de comportamento, o Elasticsearch é uma boa pedida".*

Mas o que são esses padrões de comportamento? É o que vamos descobrir no decorrer da nossa jornada! Agora, vamos iniciar o nosso percurso pelo início: instalando o ferramental.

## 1.2 INSTALAÇÃO

A instalação do Elasticsearch é muito tranquila, bastando seguir as instruções contidas na página de downloads em https://www.elastic.co/downloads/elasticsearch:

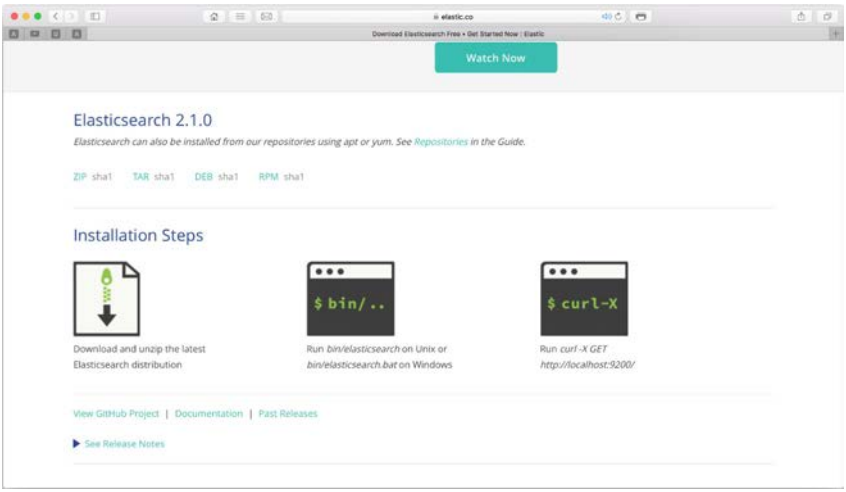



Figura 1.3: Página de download do Elasticsearch

Após desempacotar o arquivo compactado em uma pasta de sua preferência, basta entrar em um terminal na pasta descompactada e digitar:

```
bin/elasticsearch
```

Poderemos ver algumas saídas de log, e uma linha com a palavra `started` significa que subimos o nosso nó. Uma curiosidade que podemos ver é o nome do nó, que aleatoriamente será diferente a cada vez que reiniciarmos o servidor. Esses nomes são brincadeiras com nomes de personagens da Marvel, provando que, no mundo de TI, o senso de humor sempre está presente.

```
elasticsearch-2.1.0 — bin/elasticsearch — bin/elasticsearch — java -Xms256m...
[2015-11-27 19:31:38,593][INFO ][plugins                    ] [Penance] loaded [],
 sites []
[2015-11-27 19:31:38,618][INFO ][env                        ] [Penance] using [1]
data paths, mounts [[/ (/dev/disk1)]], net usable_space [52gb], net total_space
[157.3gb], spins? [unknown], types [hfs]
[2015-11-27 19:31:40,054][INFO ][node                       ] [Penance] initialize
d
[2015-11-27 19:31:40,054][INFO ][node                       ] [Penance] starting .
..
[2015-11-27 19:31:40,132][INFO ][transport                  ] [Penance] publish_ad
dress {127.0.0.1:9300}, bound_addresses {127.0.0.1:9300}, {[fe80::1]:9300}, {[::
1]:9300}
[2015-11-27 19:31:40,138][INFO ][discovery                  ] [Penance] elasticsea
rch/nQmmnq_1TTOlL6Kx89PNOg
[2015-11-27 19:31:43,170][INFO ][cluster.service            ] [Penance] new_master
 {Penance}{nQmmnq_1TTOlL6Kx89PNOg}{127.0.0.1}{127.0.0.1:9300}, reason: zen-disco
-join(elected_as_master, [0] joins received)
[2015-11-27 19:31:43,183][INFO ][http                       ] [Penance] publish_ad
dress {127.0.0.1:9200}, bound_addresses {127.0.0.1:9200}, {[fe80::1]:9200}, {[::
1]:9200}
[2015-11-27 19:31:43,183][INFO ][node                       ] [Penance] started
[2015-11-27 19:31:43,203][INFO ][gateway                    ] [Penance] recovered
[0] indices into cluster_state
```

Figura 1.4: Console do Elasticsearch

Para testar a sua instalação, abra outra janela de terminal e digite:

```
curl -X GET http://localhost:9200/
```

Esse comando retornará uma estrutura JSON informando alguns dados do cluster, como a estrutura a seguir:

```
{
  "name" : "Silver",
  "cluster_name" : "elasticsearch_alexandrelourenco",
  "version" : {
    "number" : "2.1.0",
    "build_hash" : "72cd1f1a3eee09505e036106146dc1949dc5dc87",
    "build_timestamp" : "2015-11-18T22:40:03Z",
    "build_snapshot" : false,
    "lucene_version" : "5.3.1"
  },
  "tagline" : "You Know, for Search"
}
```

Pronto! Já temos um nó de Elasticsearch operando! Vamos agora baixar as outras ferramentas.

Vamos continuar baixando agora o logstash. Para isso, entre na URL https://www.elastic.co/downloads/logstash, e repita o mesmo procedimento de desempacotamento do arquivo de instalação. Para o logstash, porém, nosso teste de instalação é ligeiramente mais complexo. Vamos navegar até a pasta `bin` da instalação do logstash e digitaremos o seguinte comando:

```
./logstash -e 'input { stdin { } } output { stdout {} }'
```

Após a mensagem `Logstash startup completed`, se digitarmos algo no console e pressionarmos `Enter`, veremos uma mensagem de "eco" da mensagem recém-digitada, como podemos ver a seguir:

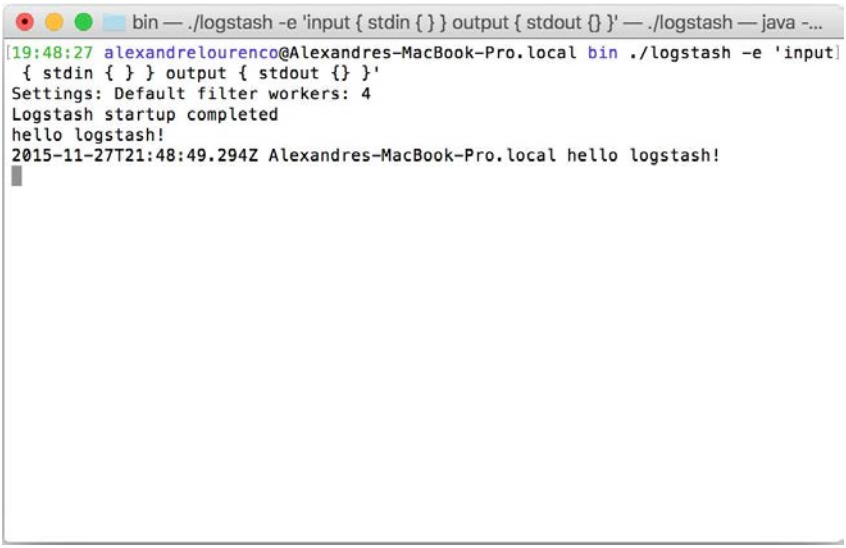



Figura 1.5: Logstash em ação

Dentro da stack ELK, o papel do Logstash pode ser considerado como uma espécie de integrador, que permite ler streams de dados de diferentes origens para diferentes destinos – não apenas instâncias de Elasticsearch. Isso permite ainda filtragens e transformações dos dados transmitidos. No próximo capítulo,

veremos mais do Logstash em ação.

Falta, por fim, instalarmos o Kibana, porém só vamos utilizá-lo daqui a alguns capítulos. Então, vamos deixar a sua instalação para o seu respectivo capítulo. Assim, concluímos a nossa instalação. Avancemos agora para o próximo capítulo, onde vamos começar explorando o Logstash.

Capítulo 2

# DISSECANDO A ELK – LOGSTASH

## 2.1 CRIANDO PIPELINES DE DADOS

Figura 2.1: Logo do Logstash

Criado pela Elastic, o conceito do Logstash é fornecer *pipelines* de dados, através do qual podemos suprir as informações contidas nos arquivos de logs das nossas aplicações – além de outras fontes – para diversos destinos, como uma instância de Elasticsearch, um *bucket* S3 na Amazon, um banco de dados MongoDB, entre outros. Todas essas capacidades são fornecidas por meio de plugins desenvolvidos pela comunidade, que já contam com mais de 165 deles.

O diagrama a seguir ilustra algumas diferentes entradas e saídas de um pipe de logstash:

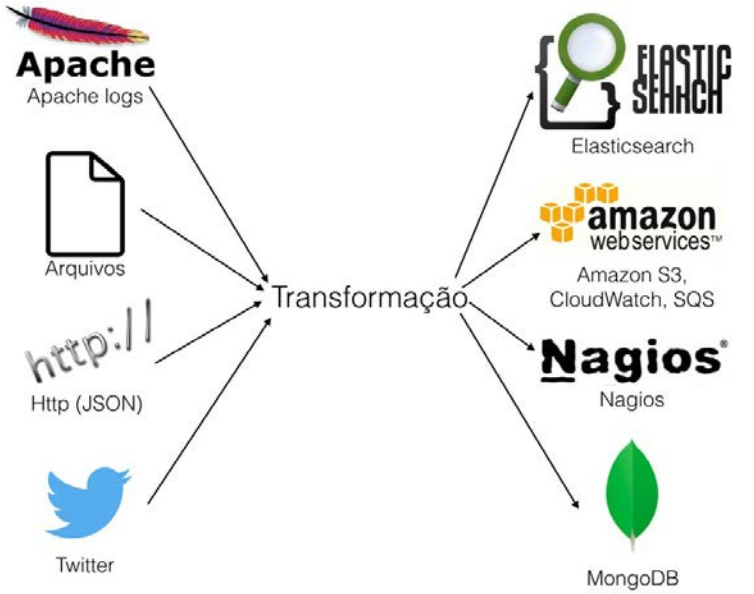



Figura 2.2: Visão macro de um pipeline logstash

Como podemos ver, o logstash se trata de uma ferramenta realmente poderosa, permitindo trabalhar com os mais diversos tipos de informações, de arquivos de logs, registros em uma coleção MongoDB, e até tweets!

Neste capítulo, vamos evoluir o nosso exemplo do capítulo anterior, incluindo mais 1 API REST e realizando as configurações necessárias para que as nossas APIs tenham seus logs integrados ao Elasticsearch. Também demonstrarei outros tipos de uso para o Logstash, de modo que você possa ter uma boa ideia do poder da ferramenta.

## 2.2 CONSTRUINDO NOSSA API DE PEDIDOS

Para começar, vamos criar outra API REST, desta vez representando uma interface de pedidos. Nesta interface, vamos

simular o funcionamento de um carrinho de compras, onde são realizadas todas as operações, como incluir e remover itens no carrinho etc. Todas essas ações são registradas também no log, para efeito de rastreabilidade.

Neste exemplo, usaremos novamente o Spring Boot. Analogamente ao capítulo anterior, o código-fonte dessa API também se encontra em meu repositório, na pasta `capítulo 2`. Nesta API, as classes `Application` e `ApplicationConfig` que criamos anteriormente possuem as mesmas configurações, bem como o restante delas. Portanto, elas serão omitidas aqui.

Vamos então começar a dissecar a classe. Começamos com os métodos de consulta que retornam todos os pedidos de um determinado cliente e todos os pedidos cadastrados:

```
    @GET
    @Produces(MediaType.APPLICATION_JSON)
    public List<Pedido> buscarPedidos() {

        logger.info("foram buscados todos os pedidos!");

        return pedidosMock;

    }

    @GET
    @Path("pedido/{idCliente}")
    @Produces(MediaType.APPLICATION_JSON)
    public List<Pedido> buscarPedidosPorCliente(@PathParam("idClie
nte") long idCliente) {

        List<Pedido> pedidos = new ArrayList<Pedido>();

        for (Pedido pedido : pedidosMock) {

            if (pedido.getIdCliente() == idCliente)
                pedidos.add(pedido);
        }

        logger.info("cliente " + idCliente + " possui " + pedidos.
size() + " pedidos");
```

```
        return pedidos;

    }
```

A seguir, criamos dois métodos responsáveis por incluir/remover itens de um pedido. Caso o pedido não exista, um novo pedido é criado no método de inclusão do item:

```
      @POST
    @Path("item/adiciona")
    @Consumes(MediaType.APPLICATION_JSON)
    public void adicionaItemPedido(ItemPedidoDTO item) {

        contadorErroCaotico++;

        if ((contadorErroCaotico) % 7 == 0) {
            throw new RuntimeException("Ocorreu um erro caótico!")
;
        }

        // se for pedido novo, cria, se não somente adiciona o item


        long idCliente = 0;

        boolean pedidoNovo = true;

        for (Pedido pedido : pedidosMock) {

            if (pedido.getId() == item.getIdPedido()) {
                pedido.getItems().add(item.getItem());

                idCliente = pedido.getIdCliente();

                pedidoNovo = false;
            }

        }

        if (pedidoNovo) {
            Pedido pedido = new Pedido();

            idCliente = item.getIdCliente();
            pedido.setId(item.getIdPedido());
            pedido.setDataPedido(new Date());
```

```
            pedido.setIdCliente(item.getIdCliente());
            pedido.getItems().add(item.getItem());
            pedido.setStatus(StatusPedido.ABERTO);

            pedidosMock.add(pedido);

        }

        logger.info("pedido " + item.getIdPedido() + " do cliente '
+ idCliente + " adicionou o produto "
                + item.getItem().getIdProduto());

    }

    @POST
    @Path("item/remove")
    @Consumes(MediaType.APPLICATION_JSON)
    public void removeItemPedido(ItemPedidoDTO item) {

        long idCliente = 0;

        for (Pedido pedido : pedidosMock) {

            if (pedido.getId() == item.getIdPedido()) {

                pedido.getItems().remove(item.getItem());

                idCliente = pedido.getIdCliente();

            }

        }

        logger.info("pedido " + item.getIdPedido() + " do cliente '
+ idCliente + " removeu o produto "
                + item.getItem().getIdProduto());

    }
```

Por fim, temos os métodos responsáveis pelo pagamento (efetivação) e cancelamento de pedidos:

```
    @PUT
    @Path("pedido/{idPedido}")
    public void pagaPedido(@PathParam("idPedido") long idPedido) {

        for (Pedido pedido : pedidosMock) {
```

```
            if (pedido.getId() == idPedido) {

                pedido.setStatus(StatusPedido.CONCLUIDO);

            }

        }

        logger.info("pedido " + idPedido + " efetivado");

    }

    @DELETE
    @Path("pedido/{idPedido}")
    public void cancelaPedido(@PathParam("idPedido") long idPedido
) {

        for (Pedido pedido : pedidosMock) {

            if (pedido.getId() == idPedido) {

                pedido.setStatus(StatusPedido.CANCELADO);

            }

        }

        logger.info("pedido " + idPedido + " cancelado");

    }
`
```

Analogamente à nossa API do capítulo passado, esta API também utiliza a biblioteca log4j2, gerando não só registros de logs no console do Spring boot como também o arquivo de log `output.log`.

Vamos começar agora com algumas chamadas, simulando o comportamento da API no mundo real. Imaginemos que um cliente deseje incluir um item no carrinho de compras e, para isso, ele faria a seguinte chamada:

```
curl -H "Content-Type: application/json" -X POST -d '{
```

```
    "idPedido": 1,
    "idCliente": 1,
    "item" : {
        "idProduto":1,
        "quantidade":1
    }

}' http://localhost:8080/item/adiciona
```

Após executar o comando, poderemos ver que no log do nosso console foi impressa uma linha como a seguinte:

```
23:40:49.510 [http-nio-8080-exec-1] INFO  br.com.alexandreesl.hand
son.rest.PedidoRestService - pedido 1 do cliente 1 adicionou o pro
duto 1
```

A seguir, vamos supor que o cliente está na tela de checkout do seu pedido e, após fornecer os dados de pagamento e ser autorizado pelo banco, chegou o momento de atualizar o status do pedido como pago. Para isso, seria feita a seguinte chamada na API:

```
curl -X PUT http://localhost:8080/pedido/1
```

Que, por sua vez, gerou uma linha de log como a seguinte:

```
23:48:11.819 [http-nio-8080-exec-3] INFO  br.com.alexandreesl.hand
son.rest.PedidoRestService - pedido 1 efetivado
```

Isso conclui os nossos testes – e código Java! – do nosso livro. Agora, começaremos o nosso foco principal, que é trabalhar com o Elasticsearch e sua stack mais famosa, o ELK.

## 2.3 COMEÇANDO COM O LOGSTASH

Recapitulando o capítulo anterior, vimos o modo mais simples de criar um pipeline logstash, que simplesmente imprime os dados recebidos pelo input do usuário no teclado:

```
./logstash -e 'input { stdin { } } output { stdout {} }'
```

Agora, vamos começar a nossa configuração das APIs, de modo

que possamos extrair os dados de seus logs usando o Logstash. Existem três modelos que podemos adotar para este pipeline:

- Utilizando o plugin **file**, podemos configurar o Logstash para realizar `poolings` em nosso arquivo `output.log`, de modo que os logs sejam incrementados em baixíssima periodicidade.

- Usando o plugin de **log4j** do Logstash, onde ele atua de modo passivo, recebendo os logs que as próprias aplicações vão transmitindo durante as suas execuções, de modo que o recebimento dos logs é quase que simultâneo à sua própria geração. Essa opção funciona muito bem no log4j1, porém, em meus estudos, encontrei alguns problemas com a conexão socket entre o Logstash e o log4j2, sendo portanto uma opção que deve ser avaliada com cuidado.

- Utilizando o plugin **gelf**. O Gelf consiste em uma solução de logging escalável, que permite inserir um servidor de logging, chamado *graylog*, como camada integradora entre as aplicações geradoras de informação e um cluster Elasticsearch, além de outros sistemas, podendo até mesmo substituir os papéis do Logstash e do Kibana dentro da stack ELK. Neste cenário, também temos o recebimento das informações quase simultaneamente à sua geração.

Para o case do livro, usarei a terceira opção, não somente pela questão de obter mais agilidade no recebimento das informações, mas também pela praticidade que o plugin nos traz, além de permitir que tenhamos múltiplas aplicações fornecendo seus logs para um único pipeline centralizado.

Porém, apesar de utilizarmos o plugin do Gelf, não usaremos o

servidor Graylog, mas sim o seu *appender* de log4j2, que se comunicará com o plugin gelf do Logstash, mantendo assim o fluxo clássico de integração da stack. O objetivo de tal estratégia é não desviar o foco do nosso estudo para outras soluções que merecem um estudo próprio em separado, além de manter a simplicidade da arquitetura proposta.

Entretanto, não desmereço a proposta do Gelf, que pode ser interessante principalmente em uma evolução futura de nosso sistema, onde o Elasticsearch se torne um importante *core* da nossa empresa, com fortíssima demanda de performance e escalabilidade. Você pode encontrar mais informações sobre essa arquitetura em https://www.graylog.org/architecture/.

Para começar, vamos montar um pipeline que recebe os logs e printa-os no console. Nós poderíamos fazer essa configuração diretamente na linha de comando, porém essa não é uma opção tão organizada. Então, vamos criar um arquivo de configuração chamado `meupipeline.conf` dentro da pasta `bin` da nossa instalação do Logstash, ou em uma outra pasta da sua preferência.

Dentro do arquivo, criaremos a seguinte configuração, que nada mais é do que uma estrutura JSON, onde temos um objeto de `input` no qual configuramos o plugin do Gelf vazio – pois usaremos todas as suas configurações default –, que define a forma como o Logstash receberá as mensagens. O resultado dessa configuração será a entrada de registros JSON, cada um representando uma linha de log.

Dentro do objeto `output`, utilizamos um simples plugin de debug, que vai printar as mensagens de logs recebidas no console. Desse modo, ficamos com a seguinte configuração:

```
input {
  gelf{
```

```
    }
}

 output {
      stdout { codec => rubydebug }
}
```

A título de curiosidade, a despeito de termos usado todas as configurações default do plugin de `input`, poderíamos realizar vários tipos de configurações, além das básicas, como por exemplo, a porta onde o processo do logstash vai escutar requisições, o charset das mensagens, tags para filtragens posteriores etc.

Agora, para executar o pipeline, basta executarmos o seguinte comando:

```
./logstash -f "<caminho para o arquivo>/meupipeline.conf"
```

Após a execução do comando, saberemos que a inicialização foi um sucesso quando uma mensagem `Logstash startup completed` for exibida, como na figura a seguir:

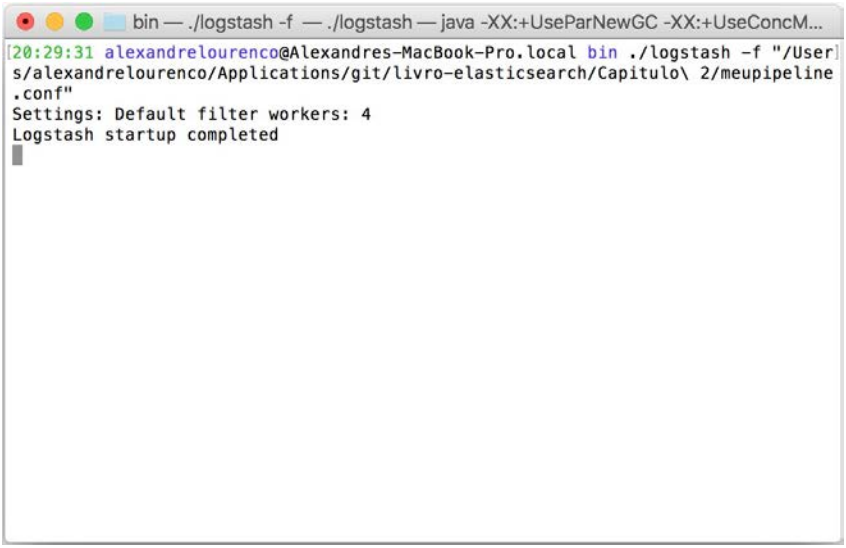



Figura 2.3: Console logstash inicializado

Agora que já configuramos o logstash, vamos configurar nossas aplicações para enviar as mensagens para ele. Para começar, vamos até os arquivos de configuração do log4j2, chamados `log4j2.xml` em ambas as APIs que já implementamos e modificamos para a configuração seguinte. Lembre-se de que todo código já se encontra disponível em meu repositório, em https://github.com/alexandreesl/livro-Elasticsearch.

```
<?xml version="1.0" encoding="UTF-8"?>
<Configuration status="WARN">
    <Appenders>
        <Console name="Console" target="SYSTEM_OUT">
            <PatternLayout pattern="%d{HH:mm:ss.SSS} [%t] %-5level
 %logger{36} - %msg%n" />
        </Console>
        <File name="File" fileName="output.log" immediateFlush="tr
ue"
            append="false">
            <PatternLayout
                pattern="%d{yyy-MM-dd HH:mm:ss.SSS} [%t] %-5level
%logger{36} - %msg%n" />
        </File>
        <Gelf name="Gelf" host="udp:localhost" port="12201" versio
="1.1" extractStackTrace="true"
              filterStackTrace="false"
              originHost="%host{fqdn}">
            <Field name="timestamp" pattern="%d{dd MMM yyyy HH:mm:
ss,SSS}" />
            <Field name="level" pattern="%level" />
            <Field name="simpleClassName" pattern="%C{1}" />
            <Field name="className" pattern="%C" />
            <Field name="server" pattern="%host" />

        </Gelf>
    </Appenders>
    <Loggers>
        <Root level="info">
            <AppenderRef ref="Console" />
            <AppenderRef ref="File" />
            <AppenderRef ref="Gelf" />
        </Root>
    </Loggers>
</Configuration>
```

Como podemos ver, definimos um `appender` de Gelf, onde

configuramos propriedades como o host e a porta para onde as mensagens serão transmitidas, o protocolo (UDP), além dos campos que queremos receber e da mensagem propriamente dita, como o stack trace em caso de erros, o timestamp de cada registro gerado no log, entre outras informações.

Nossa próxima alteração será incluir no arquivo `pom.xml` a dependência para o `jar` que contém o `appender` responsável pela integração com o Logstash. Para isso, editamos o arquivo incluindo o seguinte trecho:

```
<dependency>
     <groupId>biz.paluch.logging</groupId>
     <artifactId>logstash-gelf</artifactId>
     <version>1.7.0</version>
 </dependency>
```

Para testarmos, vamos novamente iniciar a nossa API de pedidos. Ao subirmos o Spring Boot, se observarmos a console do Logstash, já veremos as mensagens de log chegando:

```
bin — ./logstash -f — ./logstash — java — 80×24
            "message" => "Tomcat started on port(s): 8080/http",
             "server" => "alexandres-macbook-pro.local",
        "server.fqdn" => "alexandres-macbook-pro.local",
          "timestamp" => "13 Oct 2015 20:20:55,390",
          "className" => "org.springframework.boot.context.embedded.tomcat.Tomca
tEmbeddedServletContainer",
    "simpleClassName" => "TomcatEmbeddedServletContainer"
}
{
              "level" => "INFO",
               "host" => "alexandres-macbook-pro.local",
           "facility" => "logstash-gelf",
           "@version" => "1",
         "@timestamp" => "2015-10-13T23:20:55.391Z",
        "source_host" => "127.0.0.1",
            "message" => "Started Application in 2.332 seconds (JVM running for
7.943)",
             "server" => "alexandres-macbook-pro.local",
        "server.fqdn" => "alexandres-macbook-pro.local",
          "timestamp" => "13 Oct 2015 20:20:55,392",
          "className" => "org.springframework.boot.StartupInfoLogger",
    "simpleClassName" => "StartupInfoLogger"
}
```

Figura 2.4: Registros de log na console do Logstash

Agora que temos nossa API de pedidos integrada ao Logstash, vamos testar uma chamada. Para isso, chamamos novamente o comando `curl` para inserir um pedido:

```
curl -H "Content-Type: application/json" -X POST -d '{
    "idPedido": 1,
    "idCliente": 1,
    "item" : {
        "idProduto":1,
        "quantidade":1
    }

}' http://localhost:8080/item/adiciona
```

Observemos o resultado na nossa console do Logstash. Podemos ver que a nossa chamada gerou uma estrutura JSON, encapsulando os dados de log da nossa chamada.

Agora vamos testar a alteração na nossa API de Clientes. Para isso, faremos as mesmas alterações nos arquivos `log4j2.xml` e `pom.xml` que fizemos anteriormente na nossa API de Pedidos, e subiremos a nossa API de Clientes. A seguir, para testarmos que nosso Logstash está recebendo também as informações referentes a essa API, vamos pegar emprestada a chamada de inclusão de clientes do capítulo anterior, executando o seguinte comando:

```
curl http://localhost:8080/  -H "Content-Type: application/json" -
X POST -d '{"id":6,"nome":"Cliente 6","email":"customer6@gmail.com
"}'
```

Essa chamada gerará outra estrutura JSON, como podemos ver na figura:

```
bin — ./logstash -f — ./logstash — java — 80×24
        "source_host" => "127.0.0.1",
            "message" => "Started Application in 2.021 seconds (JVM running for
8.221)",
             "server" => "alexandres-macbook-pro.local",
        "server.fqdn" => "alexandres-macbook-pro.local",
          "timestamp" => "13 Oct 2015 21:27:37,743",
          "className" => "org.springframework.boot.StartupInfoLogger",
    "simpleClassName" => "StartupInfoLogger"
}
{
              "level" => "WARN",
               "host" => "alexandres-macbook-pro.local",
           "facility" => "logstash-gelf",
           "@version" => "1",
         "@timestamp" => "2015-10-14T00:28:06.242Z",
        "source_host" => "127.0.0.1",
            "message" => "O cliente 6 foi inserido!",
             "server" => "alexandres-macbook-pro.local",
        "server.fqdn" => "alexandres-macbook-pro.local",
          "timestamp" => "13 Oct 2015 21:28:06,242",
          "className" => "br.com.alexandreesl.handson.rest.ClienteRestService",
    "simpleClassName" => "ClienteRestService"
}
```

Figura 2.5: Resultado da chamada da inclusão de clientes na console do Logstash

Perfeito! Agora temos um pipeline do Logstash que recebe mensagens de nossas APIs e as imprime na tela.

Você poderá notar que, dentro do nosso JSON, temos a mensagem que montamos nas nossas chamadas, a classe `logger` escrita por inteiro, como um único campo. Dentro dessa mensagem, podemos ver diversas informações que podem ser interessantes para nós futuramente, como os códigos dos clientes que estão sendo inseridos, por exemplo.

Seria muito interessante se pudéssemos ter essas informações "destrinchadas" separadamente, de modo que pudéssemos trabalhar esses dados. Isso é precisamente o que vamos fazer agora, usando o plugin de transformação – ou filtragem – chamado Grok.

## 2.4 PARSEANDO AS INFORMAÇÕES DE LOG

O plugin **grok** foi criado com o objetivo de obter dados textuais

de uma fonte e parseá-los em uma nova estrutura. Vamos usá-lo para parsear o nosso campo `message`, a fim de obter informações mais apuradas dos nossos logs.

O Grok provê uma grande série de expressões regulares já prontas que podemos utilizar em nossos logs, como podemos ver no fragmento adiante. Para ver a lista completa, o leitor pode acessar o endereço https://github.com/elastic/logstash/blob/v1.4.2/patterns/grok-patterns.

```
USERNAME [a-zA-Z0-9._-]+
USER %{USERNAME}
INT (?:[+-]?(?:[0-9]+))
BASE10NUM (?<![0-9.+-])(?>[+-]?(?:(?:[0-9]+(?:\.[0-9]+)?)|(?:\.[0-9
]+)))
NUMBER (?:%{BASE10NUM})
BASE16NUM (?<![0-9A-Fa-f])(?:[+-]?(?:0x)?(?:[0-9A-Fa-f]+))
BASE16FLOAT \b(?<![0-9A-Fa-f.])(?:[+-]?(?:0x)?(?:(?:[0-9A-Fa-f]+(?
:\.[0-9A-Fa-f]*)?)|(?:\.[0-9A-Fa-f]+)))\b

POSINT \b(?:[1-9][0-9]*)\b
NONNEGINT \b(?:[0-9]+)\b
WORD \b\w+\b
NOTSPACE \S+
SPACE \s*
DATA .*?
GREEDYDATA .*
QUOTEDSTRING (?>(?<!\\)(?>"(?>\\.|[^\\"]+)+"|""|(?>'(?>\\.|[^\\']+
)+')|''|(?>`(?>\\.|[^\\`]+)+`)|``))
UUID [A-Fa-f0-9]{8}-(?:[A-Fa-f0-9]{4}-){3}[A-Fa-f0-9]{12}
... RESTANTE OMITIDO ...
```

Durante o nosso desenvolvimento do parse, uma boa dica é utilizar o site https://grokdebug.herokuapp.com, que permite que coloquemos nossas linhas de mensagens de logs e testemos a montagem do nosso parse em tempo real, em um modelo REPL (*Read-Eval-Process-Loop*).

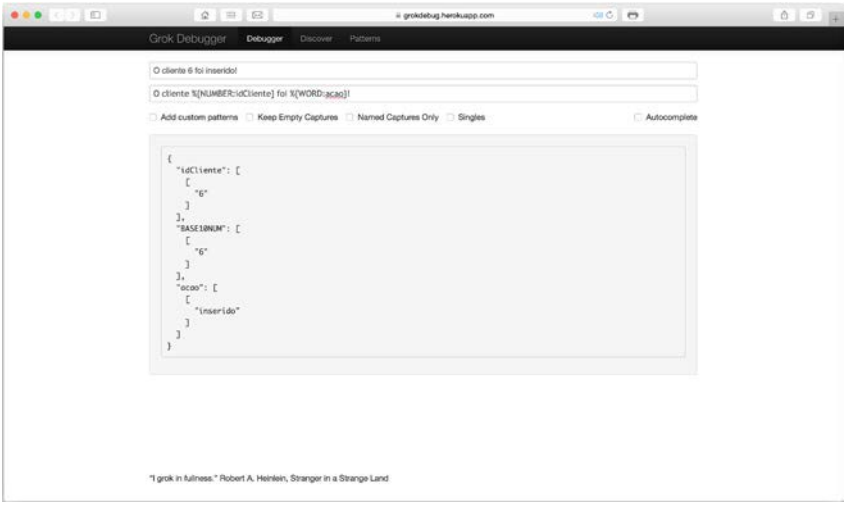



Figura 2.6: Utilizando o logstash para parsear os clientes

Nessa figura, podemos ver o primeiro dos nossos parses, que realizará o parse das informações de clientes. Para o configurarmos, modificamos o arquivo `meupipeline.conf` para incluir o plugin e a expressão de parse:

```
input {
  gelf{

    }
}

filter {
grok {
match => [ "message" , "O cliente %{NUMBER:idCliente} foi %{WORD:a
cao}!" ,
            "message" , "%{GREEDYDATA:logdefault}" ]
}
}

 output {
      stdout { codec => rubydebug }
}
```

Você vai notar que também criamos na listagem de expressões um parse que consiste de apenas um campo do tipo `GREEDYDATA` ,

que criará um campo chamado `logdefault` . O objetivo dessa expressão é prover um parse default para as mensagens recebidas pelo Logstash, para os casos em que não desejamos nenhuma filtragem especial.

A ordem de avaliação do plugin é sempre da esquerda para a direita, ou seja, se a mensagem tiver o seu parse avaliado e for detectado que esta faz match com a primeira expressão, ela será aplicada; caso contrário, passará para a próxima expressão e assim por diante, até que a última seja avaliada. Se nenhuma das expressões avaliadas fizer match nas avaliações, a mensagem não será descartada, porém uma tag indicando a falha do plugin será incluída para indicar o problema.

Um ponto importante a se destacar nessa ordem de avaliação é que devemos ficar atentos para quando criarmos expressões que sejam contidas umas nas outras, isto é, expressões que podem ser "encaixadas" uma dentro da outra. Nesta configuração mesmo, temos um exemplo dessa situação, que é a expressão que deixamos por último, que gera o campo `logdefault` . Ela faz match com toda e qualquer mensagem que passe por ela, ou seja, se a colocarmos como primeiro da listagem, ela sempre fará match no pipeline, ocasionando que o nosso parse de clientes nunca seja executado!

Após a alteração, vamos reiniciar o nosso pipeline – pressionando `Crtl-C` na janela do terminal e reexecutando o comando –, e vamos novamente executar o comando `curl` que testamos a pouco. Poderemos perceber na nova mensagem gerada após a chamada de inclusão do cliente que dois novos campos foram criados, um chamado `idCliente` e um chamado `acao` , que nos permitem trabalhar os dados de clientes que foram inseridos e alterados:

```
bin — ./logstash -f — ./logstash — java — 80×24
        "server.fqdn" => "Alexandres-MacBook-Pro.local",
          "timestamp" => "13 Oct 2015 22:38:09,371",
          "className" => "org.springframework.boot.StartupInfoLogger",
    "simpleClassName" => "StartupInfoLogger",
         "logdefault" => "Started Application in 1.935 seconds (JVM running for
12.53)"
}
{
              "level" => "WARN",
               "host" => "Alexandres-MacBook-Pro.local",
           "facility" => "logstash-gelf",
           "@version" => "1",
         "@timestamp" => "2015-10-14T01:38:30.917Z",
        "source_host" => "127.0.0.1",
            "message" => "O cliente 6 foi inserido!",
             "server" => "Alexandres-MacBook-Pro.local",
        "server.fqdn" => "Alexandres-MacBook-Pro.local",
          "timestamp" => "13 Oct 2015 22:38:30,918",
          "className" => "br.com.alexandreesl.handson.rest.ClienteRestService",
    "simpleClassName" => "ClienteRestService",
           "idCliente" => "6",
                "acao" => "inserido"
}
```

Figura 2.7: Utilizando o grok para parsear as informações de clientes

Nosso parse é um sucesso! O leitor pode notar que deixamos de fora alguns tipos de mensagens, como a de consulta de clientes. A razão para isso é simplesmente porque, para o que planejamos construir, não precisamos parsear aqueles tipos de mensagens. Deixo um desafio para o leitor concluir a captura das informações também para as consultas dos clientes.

Agora, vamos passar para o parse das informações de pedidos. Para essa API, deixaremos de fora apenas a consulta de todos os pedidos, construindo diversas expressões de parseamento das diferentes informações. Vamos seguir a ordem de implementação dos métodos, começando pela busca de pedidos por cliente. Nessa consulta, que efetuamos com o comando:

```
curl -X GET http://localhost:8080/pedido/1
```

Obtemos mensagens de log como:

```
20:28:41.692 [http-nio-8080-exec-7] INFO  br.com.alexandreesl.hand
son.rest.PedidoRestService - cliente 1 possui 1 pedidos
```

Para obtermos dessa consulta a quantidade de pedidos que cada cliente possui na data e hora em que a consulta foi efetuada, vamos construir o seguinte parse:

```
cliente %{NUMBER:idCliente} possui %{NUMBER:qtdPedidos} pedidos
```

Você pode estar se perguntando: *"Mas e a data e hora?"*. Se repararmos no JSON gerado na console do Logstash, veremos que já existe um campo de data e hora que é gerado pelo nosso appender log4j2. Sendo assim, não precisamos de maiores tratativas para esta questão.

A seguir, temos os métodos de inclusão e exclusão de itens de um pedido. Para esses métodos, criaremos um único parse, que é o seguinte:

```
pedido %{NUMBER:idPedido} do cliente %{NUMBER:idCliente} %{WORD:ac
aoItemPedido} o produto %{NUMBER:idProdutoPedido}
```

> Aqui vale um ponto de atenção: você pode ter reparado que usamos como nome do campo da ação realizada no pedido o valor `acaoItemPedido`. O leitor pode ficar tentado a utilizar simplesmente o valor `acao`, porém, se você observar as expressões que criamos neste capítulo, notará que já criamos um campo com esse nome no parse da API de clientes.
>
> Entenderemos com mais detalhes no próximo capítulo a estrutura do Elasticsearch, entretanto, por hora, basta sabermos que temos de tomar muito cuidado com os nomes dos campos que parseamos com o Grok, para que eles não se misturem.

Por fim, temos os métodos de efetivação e cancelamento de pedidos, nos quais também resolveremos o mesmo parseamento

com uma única expressão:

```
pedido %{NUMBER:idPedido} %{WORD:acaoPedido}
```

Assim, concluindo o trabalho, vamos modificar novamente nosso arquivo de configuração do pipeline, adicionando as nossas novas expressões:

```
input {
  gelf{

    }
}

filter {
grok {
match => [ "message" , "O cliente %{NUMBER:idCliente} foi %{WORD:a
cao}!" ,
            "message", "cliente %{NUMBER:idCliente} possui %{NUMBE
R:qtdPedidos} pedidos",
            "message", "pedido %{NUMBER:idPedido} do cliente %{NUM
BER:idCliente} %{WORD:acaoItemPedido} o produto %{NUMBER:idProduto
Pedido}",
            "message", "pedido %{NUMBER:idPedido} %{WORD:acaoPedid
o}",
            "message" , "%{GREEDYDATA:logdefault}" ]
}
}

 output {
      stdout { codec => rubydebug }
}
```

Agora, vamos testar a nossa implementação. Para começar, novamente reiniciaremos o pipeline e, com nossas APIs devidamente inicializadas, começamos por executar novamente uma inclusão de cliente. Se o leitor tomar um erro quando inicializar a segunda API, é porque o Spring Boot sempre tenta subir na porta 8080 por padrão. Para trocar a porta, basta incluir no comando de start o parâmetro `-Dserver.port=<nova porta>`.

```
curl http://localhost:8080/  -H "Content-Type: application/json" -
X POST -d '{"id":6,"nome":"Cliente 6","email":"customer6@gmail.com
```

```
"}'
```

Como resultado, teremos o seguinte JSON na nossa console do Logstash:

```
{
              "level" => "WARN",
               "host" => "alexandres-macbook-pro.local",
           "facility" => "logstash-gelf",
           "@version" => "1",
         "@timestamp" => "2015-10-15T00:56:50.051Z",
        "source_host" => "127.0.0.1",
            "message" => "O cliente 6 foi inserido!",
             "server" => "alexandres-macbook-pro.local",
        "server.fqdn" => "alexandres-macbook-pro.local",
          "timestamp" => "14 Oct 2015 21:56:50,051",
          "className" => "br.com.alexandreesl.handson.rest.Cliente
RestService",
    "simpleClassName" => "ClienteRestService",
          "idCliente" => "6",
               "acao" => "inserido"
}
```

A seguir, testamos inserir 2 itens em um pedido, que, por ser novo, consequentemente será criado, com a sequência de comandos:

```
curl -H "Content-Type: application/json" -X POST -d '{
    "idPedido": 1,
    "idCliente": 1,
    "item" : {
        "idProduto":1,
        "quantidade":1
    }

}' http://localhost:8080/item/adiciona

curl -H "Content-Type: application/json" -X POST -d '{
    "idPedido": 1,
    "item" : {
        "idProduto":2,
        "quantidade":3
    }

}' http://localhost:8080/item/adiciona
```

Eles geram, respectivamente, os JSONs:

```
{
              "level" => "INFO",
               "host" => "alexandres-macbook-pro.local",
           "facility" => "logstash-gelf",
           "@version" => "1",
         "@timestamp" => "2015-10-27T00:13:37.056Z",
        "source_host" => "127.0.0.1",
            "message" => "pedido 1 do cliente 1 adicionou o produt
o 1",
             "server" => "alexandres-macbook-pro.local",
        "server.fqdn" => "alexandres-macbook-pro.local",
          "timestamp" => "26 Oct 2015 22:13:37,057",
          "className" => "br.com.alexandreesl.handson.rest.PedidoR
estService",
    "simpleClassName" => "PedidoRestService",
           "idPedido" => "1",
          "idCliente" => "1",
     "acaoItemPedido" => "adicionou",
    "idProdutoPedido" => "1"
}
```

```
{
              "level" => "INFO",
               "host" => "alexandres-macbook-pro.local",
           "facility" => "logstash-gelf",
           "@version" => "1",
         "@timestamp" => "2015-10-27T00:30:21.865Z",
        "source_host" => "127.0.0.1",
            "message" => "pedido 1 do cliente 1 adicionou o produt
o 2",
             "server" => "alexandres-macbook-pro.local",
        "server.fqdn" => "alexandres-macbook-pro.local",
          "timestamp" => "26 Oct 2015 22:30:21,865",
          "className" => "br.com.alexandreesl.handson.rest.PedidoR
estService",
    "simpleClassName" => "PedidoRestService",
           "idPedido" => "1",
          "idCliente" => "1",
     "acaoItemPedido" => "adicionou",
    "idProdutoPedido" => "2"
}
```

Por fim, testamos efetivar o nosso pedido pelo comando:

```
curl -X PUT http://localhost:8080/pedido/1
```

Ele gerará um JSON como o seguinte:

```
{
              "level" => "INFO",
               "host" => "alexandres-macbook-pro.local",
           "facility" => "logstash-gelf",
           "@version" => "1",
         "@timestamp" => "2015-10-15T01:13:52.368Z",
        "source_host" => "127.0.0.1",
            "message" => "pedido 1 efetivado",
             "server" => "alexandres-macbook-pro.local",
        "server.fqdn" => "alexandres-macbook-pro.local",
          "timestamp" => "14 Oct 2015 22:13:52,369",
          "className" => "br.com.alexandreesl.handson.rest.PedidoR
estService",
    "simpleClassName" => "PedidoRestService",
           "idPedido" => "1",
         "acaoPedido" => "efetivado"
}
```

A seguir, temos o cancelamento de pedidos, que testamos com a chamada:

```
curl -X DELETE http://localhost:8080/pedido/1
```

Esta, por sua vez, gerará um JSON como:

```
{
              "level" => "INFO",
               "host" => "alexandres-macbook-pro.local",
           "facility" => "logstash-gelf",
           "@version" => "1",
         "@timestamp" => "2015-10-15T01:16:45.351Z",
        "source_host" => "127.0.0.1",
            "message" => "pedido 1 cancelado",
             "server" => "alexandres-macbook-pro.local",
        "server.fqdn" => "alexandres-macbook-pro.local",
          "timestamp" => "14 Oct 2015 22:16:45,352",
          "className" => "br.com.alexandreesl.handson.rest.PedidoR
estService",
    "simpleClassName" => "PedidoRestService",
           "idPedido" => "1",
         "acaoPedido" => "cancelado"
}
```

E finalmente, temos a consulta da quantidade de pedidos de um cliente, que testamos com a chamada:

```
curl -X GET http://localhost:8080/pedido/1
```

Que, por fim, gera um JSON como:

```
{
              "level" => "INFO",
               "host" => "alexandres-macbook-pro.local",
           "facility" => "logstash-gelf",
           "@version" => "1",
         "@timestamp" => "2015-10-15T01:23:47.861Z",
        "source_host" => "127.0.0.1",
            "message" => "cliente 1 possui 1 pedidos",
             "server" => "alexandres-macbook-pro.local",
        "server.fqdn" => "alexandres-macbook-pro.local",
          "timestamp" => "14 Oct 2015 22:23:47,861",
          "className" => "br.com.alexandreesl.handson.rest.PedidoR
estService",
    "simpleClassName" => "PedidoRestService",
          "idCliente" => "1",
         "qtdPedidos" => "1"
}
```

Sucesso! Conseguimos implementar o nosso pipeline! Com isso, concluímos a parte prática do nosso capítulo. No restante do capítulo, faremos uma breve pincelada nos conceitos e outros plugins que o Logstash possui. Portanto, se o leitor desejar ir direto para a continuação da prática no próximo capítulo e voltar posteriormente aqui quando desejar alguma consulta de caráter mais referencial com relação ao Logstash, não há problemas.

**Observação**

Na nossa prática, configuramos as expressões diretamente dentro do arquivo de configuração do Logstash. Sugiro para o leitor como lição de casa estudar a propriedade `patterns_dir`, que permite externalizar as expressões em arquivos externalizados, a fim de melhorar a organização do trabalho.

## 2.5 CONCEITOS E OUTROS PLUGINS

Conforme pudemos ver no decorrer da nossa prática, em um arquivo de configuração de pipeline temos 3 seções distintas, que são:

- `input` – Nessa seção, são configuradas as fontes de dados do pipeline, como logs, arquivos, Twitter etc. Aqui também são configurados os `codecs`, responsáveis por realizar transformações de formato dos dados de entrada, como a conversão de binário para textual. Um conhecido plugin do pessoal que usa Logstash com Java - mas que não precisamos usar aqui por conta do nosso *appender* do Gelf já realizar esse trabalho - é o *multiline*, que concatena eventos de logs multilinhas em uma única linha. Isso é necessário devido aos momentos em que o Java dispara um stack trace, que, sem o devido tratamento, gerará um evento no pipeline para cada linha do stack trace!

- `filter` – Nessa seção, são configuradas transformações mais profundas nos dados, como os parseamentos que vimos anteriormente, formatação dos dados para padrões como CSV e XML, e até mesmo a geração de *checksums* dos eventos recebidos.

- `output` – Nessa seção, por fim, configuramos as saídas dos pipelines, que dentre diversos formatos, temos o que vamos estudar no próximo capítulo, o Elasticsearch.

A seguir, detalharei os plugins que acho mais interessantes de cada seção, que vale a pena o leitor conhecer.

### Plugins de input

- **File:** permite que um pipeline efetue o pooling de pastas de `file system` ;
- **GitHub:** permite a leitura de eventos de log do GitHub;
- **Http:** fornece uma porta de escuta onde requisições HTTP POST em formato JSON podem ser recebidas;
- **Twitter:** permite montar um pipeline que recebe uma stream de tweets por meio da Twitter API;
- **S3:** análogo ao File, porém lendo os arquivos de um AWS bucket;
- **Websocket:** permite a leitura de eventos pelo protocolo websocket;
- **Sqs:** assim como o s3, também permite utilizar os recursos da AWS, porém, neste caso, de uma fila SQS;

Esses e muitos outros plugins de input podem ser encontrados em https://www.elastic.co/guide/en/logstash/current/input-plugins.html.

## Plugins de filter

- **Grok:** um dos plugins mais conhecidos, permite que campos em texto plano sejam parseados formando diferentes campos;
- **Multiline:** conforme dito anteriormente, este plugin permite que eventos que possuem múltiplas linhas sejam encapsuladas em uma única linha;
- **Cipher:** esse plugin permite que mensagens sejam criptografadas ou descriptografadas antes de serem enviadas para o destino;

- **Checksum:** gera um checksum da mensagem, permitindo assim que ela possua um identificador único;

- **I18n:** permite a remoção de caracteres especiais (como acentuação) dos eventos no pipeline;

- **Json:** plugin de missão parecida com o Grok, permite campos que possuam uma estrutura JSON em seus valores sejam manipulados, adicionando ou removendo-se campos, tags etc.;

- **Elasticsearch:** sim, ele também está no filter! Esse plugin permite que o Logstash pesquise no Elasticsearch sobre algum evento já enviado que possua informações relacionadas ao evento que se encontra naquele momento no pipeline. Caso encontre algum evento relacionado, o plugin efetua a cópia de campos pré-definidos para o evento atual antes de enviá-lo também para o Elasticsearch, permitindo assim a implementação de complexos rastreamentos.

Esses e muitos outros plugins de filter podem ser encontrados em https://www.elastic.co/guide/en/logstash/current/filter-plugins.html.

## Plugins de output

- **Elasticsearch:** plugin que permite que os eventos do pipeline sejam integrados a um cluster Elasticsearch, e o veremos no próximo capítulo;

- **File:** plugin que gera arquivos no `file system` para os eventos que passaram no pipeline;

- **Http:** permite que os eventos sejam transmitidos em

diversos formatos, como JSON, utilizando diversos HTTP METHODS, como `POST` , `GET` etc.;

- **MongoDB:** permite que eventos sejam gravados no formato de documentos em uma collection de um MongoDB;
- **Email:** permite configurar um e-mail para o qual os eventos serão enviados;
- **Google_bigquery:** permite que os eventos sejam enviados para o Google Big Query.

Esses e muitos outros plugins de output podem ser encontrados em https://www.elastic.co/guide/en/logstash/current/output-plugins.html

## 2.6 FILTROS CONDICIONAIS

Você pode estar se perguntando: *"Se a seção se chama filter, onde está a filtragem?"*. Na verdade, não precisamos usar o recurso em nossa prática, mas é possível sim efetuar filtragens nos eventos do pipeline, descartando os que não nos interessam que sejam enviados pelo output. Você pode encontrar mais informações sobre esse recurso em https://www.elastic.co/guide/en/logstash/current/config-examples.html.

## 2.7 CONCLUSÃO

E assim, concluímos o nosso capítulo sobre o Logstash. Espero ter dado uma ideia para o leitor do grande poder que temos a disposição com uma ferramenta tão simples como o Logstash. Acompanhe-me para o próximo capítulo, onde começamos nossa

conversa sobre o Elasticsearch.

CAPÍTULO 3

# DISSECANDO A ELK - ELASTICSEARCH

## 3.1 MONTANDO UM CLUSTER DE BUSCAS FULL TEXT

Figura 3.1: Logo do Elasticsearch

Continuando nossos estudos, vamos agora falar sobre a principal ferramenta abordada por este livro, o Elasticsearch. Conforme vimos anteriormente, o Elasticsearch foi criado por Shay Banon em 2010, e é baseado no Apache Lucene. Dentro do Elasticsearch, temos os conceitos de fragmentos (*shards*), documentos e índices, que vamos abordar no decorrer deste capítulo. Vamos começar revisitando a configuração de *Logstash* que fizemos no capítulo anterior, incluindo a perna que integra o Logstash ao Elasticsearch.

Todo o código prático deste capítulo está disponível em https://github.com/alexandreesl/livro-

elasticsearch/tree/master/Capitulo%203.

## 3.2 INTEGRANDO AS FERRAMENTAS

No capítulo anterior, paramos a configuração no momento em que tínhamos os dados sendo coletados das nossas APIs, porém, apenas imprimíamos os eventos de logs na console. Vamos agora modificar essa configuração para enviar nossos eventos para o Elasticsearch.

Para começar, vamos levantar a nossa instância de Elasticsearch, análogo ao que fizemos no capítulo *Introdução*, navegando até a nossa pasta de instalação e digitando o comando:

```
bin/elasticsearch
```

Agora que temos o Elasticsearch rodando, vamos alterar a nossa configuração do Logstash. Vamos modificar o nosso arquivo `meupipeline.conf` para a seguinte configuração, onde incluímos o plugin do Elasticsearch, definindo o IP, porta e o padrão de nome de índice a ser criado:

```
input {
  gelf{

    }
}

filter {
grok {
match => [ "message" , "O cliente %{NUMBER:idCliente} foi %{WORD:a
cao}!" ,
            "message", "cliente %{NUMBER:idCliente} possui %{NUMBE
R:qtdPedidos} pedidos",
            "message", "pedido %{NUMBER:idPedido} do cliente %{NUM
BER:idCliente} %{WORD:acaoItemPedido} o produto %{NUMBER:idProduto
Pedido}",
            "message", "pedido %{NUMBER:idPedido} %{WORD:acaoPedid
o}",
            "message" , "%{GREEDYDATA:logdefault}" ]
```

```
}
}

 output {
      stdout { codec => rubydebug }
      elasticsearch {
            hosts => [ "localhost:9200" ]
            index => "testes-%{+YYYY.MM.dd}"
        }
}
```

Agora, vamos reiniciar o nosso Logstash com a nossa nova configuração, e testá-la simulando novamente a ação de incluir um item ao carrinho de compras:

```
curl -H "Content-Type: application/json" -X POST -d '{
    "idPedido": 1,
    "idCliente": 1,
    "item" : {
        "idProduto":2,
        "quantidade":1
    }

}' http://localhost:8080/item/adiciona
```

O leitor pode notar que continuamos com o plugin que imprime no console ativado, apenas para efeitos de teste. Podemos ver que o pipeline continua funcionando, já que os logs foram impressos no console, mas e quanto ao Elasticsearch? Será que já temos registros inseridos nele? Vamos testar os nossos logs fazendo a nossa primeira consulta no Elasticsearch!

Vamos supor que já temos o nosso Elasticsearch operando na companhia normalmente, e desejamos pesquisar os logs pelos eventos onde os clientes adicionaram itens aos seus carrinhos de compras. Neste caso, a forma mais simples de efetuarmos essa consulta é com o comando (mais adiante no livro, veremos como construir uma interface para isso com o Kibana):

```
curl -XGET 'localhost:9200/testes-*/_search?pretty&q=acaoItemPedid
```

```
o:adicionou'
```

Quando efetuamos essa consulta, veremos que o Elasticsearch nos retorna uma estrutura JSON com o resultado da consulta, como a seguinte:


```
{
  "took" : 2,
  "timed_out" : false,
  "_shards" : {
    "total" : 5,
    "successful" : 5,
    "failed" : 0
  },
  "hits" : {
    "total" : 1,
    "max_score" : 0.30685282,
    "hits" : [ {
      "_index" : "testes-2015.10.27",
      "_type" : "logs",
      "_id" : "AVCmulZzMYVF93THYgjv",
      "_score" : 0.30685282,
      "_source":{"level":"INFO","host":"alexandres-macbook-pro.loc
al","facility":"logstash-gelf","@version":"1","@timestamp":"2015-1
0-27T00:37:21.019Z","source_host":"127.0.0.1","message":"pedido 1
do cliente 1 adicionou o produto 2","server":"alexandres-macbook-p
ro.local","server.fqdn":"alexandres-macbook-pro.local","timestamp":
"26 Oct 2015 22:37:21,020","className":"br.com.alexandreesl.handso
n.rest.PedidoRestService","simpleClassName":"PedidoRestService","i
dPedido":"1","idCliente":"1","acaoItemPedido":"adicionou","idProdu
toPedido":"2"}
    } ]
  }
}
```


Nesses resultados, além do documento – nome dado às unidades de informação indexadas seguindo a terminologia do Elasticsearch –, podemos ver que diversas informações foram impressas relacionadas à pesquisa propriamente dita, como o tempo que a consulta levou para executar pelo parâmetro `took`, o total de `shards` que a pesquisa percorreu etc. Você reparou que existem alguns campos que mapeamos no grok, como `idPedido` e o próprio `acaoItemPedido`, que usamos para fazer a consulta? Isso é

por conta do Logstash, que passou para o Elasticsearch esses campos adicionais que criamos.

Podemos identificar, porém, um problema na nossa configuração: alguns campos, como o citado `idPedido`, estão representados entre aspas duplas, o que indica que eles estão sendo armazenados como sequências de caracteres ( `strings` ) em vez de numéricos, o que seria o correto.

Vamos corrigir essa questão utilizando outro plugin chamado `mutate`, que transformará os types dos campos antes de enviar para o Elasticsearch. Além disso, vamos também trocar o padrão de nome dos índices que serão criados pelo pipeline para o nome definitivo que usaremos no nosso estudo.

Para isso, vamos modificar o arquivo `meupipeline.conf` para a seguinte configuração:

```
input {
  gelf{

    }
}

filter {
    grok {
    match => [ "message" , "O cliente %{NUMBER:idCliente} foi %{WO
RD:acao}!" ,
                "message", "cliente %{NUMBER:idCliente} possui %{N
UMBER:qtdPedidos} pedidos",
                "message", "pedido %{NUMBER:idPedido} do cliente %
{NUMBER:idCliente} %{WORD:acaoItemPedido} o produto %{NUMBER:idPro
dutoPedido}",
                "message", "pedido %{NUMBER:idPedido} %{WORD:acaoP
edido}",
                "message" , "%{GREEDYDATA:logdefault}" ]
    }
   mutate { "convert" => [  "idCliente" , "integer"  ] }
   mutate { "convert" => [  "qtdPedidos" , "integer"  ] }
   mutate { "convert" => [  "idPedido" , "integer"  ] }
   mutate { "convert" => [  "idProdutoPedido" , "integer"  ] }
```

```
}

 output {
      stdout { codec => rubydebug }
      elasticsearch {
            hosts => ["localhost:9200"]
            index => "casadocodigo-%{+YYYY.MM.dd}"
        }
}
```

E vamos reiniciar o Logstash. Simularemos outra chamada de cliente adicionando item no seu carrinho de compras e, finalmente, efetuaremos uma nova consulta no Elasticsearch, trocando apenas o índice – o tradicional contexto web dentro de uma URL – para vermos se os types foram adequadamente modificados:

```
curl -XGET 'localhost:9200/casadocodigo-*/_search?pretty&q=acaoIte
mPedido:adicionou'
```

Após a chamada, ao vermos o resultado da consulta, veremos que os campos não possuem mais aspas duplas, provando que a nossa configuração foi um sucesso:

```
{
  "took" : 2,
  "timed_out" : false,
  "_shards" : {
    "total" : 5,
    "successful" : 5,
    "failed" : 0
  },
  "hits" : {
    "total" : 5,
    "max_score" : 4.2580967,
    "hits" : [ {
      "_index" : "casadocodigo-2015.10.27",
      "_type" : "logs",
      "_id" : "AVCmuc04MYVF93THYgju",
      "_score" : 4.2580967,
      "_source":{"level":"INFO","host":"alexandres-macbook-pro.loc
al","facility":"logstash-gelf","@version":"1","@timestamp":"2015-1
0-27T00:36:46.000Z","source_host":"127.0.0.1","message":"pedido 1
do cliente 1 adicionou o produto 2","server":"alexandres-macbook-p
```

ro.local","server.fqdn":"alexandres-macbook-pro.local","timestamp":
"26 Oct 2015 22:36:46,000","className":"br.com.alexandreesl.handso
n.rest.PedidoRestService","simpleClassName":"PedidoRestService","i
dPedido":1,"idCliente":1,"acaoItemPedido":"adicionou","idProdutoPe
dido":2}
    }

**Importante**

Como vamos modificar a estrutura de armazenamento (os types) dos logs, não seria possível de maneira fácil migrar os dados do índice antigo para o novo, o que ressalta um *pitfall* com que se deve tomar todo cuidado no plano de adoção da tecnologia. Para migrações mais convencionais, é possível utilizar o mecanismo de geração e restore de snapshots do Elasticsearch, que veremos em ação no capítulo *Administrando um cluster Elasticsearch* do nosso livro.

Mas afinal, o que são índices, documentos e shards? Vamos entrar agora nos conceitos da estrutura do Elasticsearch, muito importantes para que possamos usar adequadamente o nosso motor de busca.

## 3.3 ENTENDENDO A ESTRUTURA INTERNA DO ELASTICSEARCH

Conforme dito anteriormente, o Elasticsearch é construído sob o Apache Lucene. Isso significa que, assim como no Lucene, as informações indexadas no Elasticsearch são agrupadas em índices. Fazendo uma certa analogia, imagine que um índice seria algo como um banco de dados, onde tabelas são definidas e agrupadas para posterior inserção/atualização/consulta etc.

Seguindo a nossa analogia, dentro de um banco, temos conjuntos de tabelas, que armazenam informações. Dentro do mundo do Elasticsearch, nós temos documentos, que são as estruturas onde os eventos de logs e outros tipos de informações que queremos armazenar são estruturadas e armazenadas. Você reparou no campo `type` que existe dentro das estruturas JSON que visualizamos há pouco?

Nesse campo de nome `logs`, o valor default da propriedade `type` do plugin do logstash está como o nome do nosso tipo de documento, onde as informações referentes aos logs capturados das APIs estão sendo armazenadas e indexadas. Cada documento possui o seu mapeamento, ou `document type`, que armazena as informações referentes aos campos do documento e seus respectivos tipos. Para visualizarmos o `document type` do nosso tipo `logs`, por exemplo, basta executarmos este comando `curl`:

```
curl -XGET 'localhost:9200/casadocodigo-*/_mapping?pretty'
```

Quando executamos essa consulta, recebemos de volta todos os `document types` cadastrados para aquele índice, dentro de uma estrutura JSON. Neste caso, recebemos apenas o `document type` do nosso documento `logs`, que é o único criado até este instante no índice:

```
{
  "casadocodigo-2015.10.27" : {
    "mappings" : {
      "logs" : {
        "properties" : {
          "@timestamp" : {
            "type" : "date",
            "format" : "dateOptionalTime"
          },
          "@version" : {
            "type" : "string"
          },
          "StackTrace" : {
            "type" : "string"
```

```
      },
      "acao" : {
        "type" : "string"
      },
      "acaoItemPedido" : {
        "type" : "string"
      },
      "acaoPedido" : {
        "type" : "string"
      },
      "className" : {
        "type" : "string"
      },
      "facility" : {
        "type" : "string"
      },
      "host" : {
        "type" : "string"
      },
      "idCliente" : {
        "type" : "long"
      },
      "idPedido" : {
        "type" : "long"
      },
      "idProdutoPedido" : {
        "type" : "long"
      },
      "level" : {
        "type" : "string"
      },
      "logdefault" : {
        "type" : "string"
      },
      "message" : {
        "type" : "string"
      },
      "server" : {
        "type" : "string"
      },
      "server.fqdn" : {
        "type" : "string"
      },
      "short_message" : {
        "type" : "string"
      },
      "simpleClassName" : {
        "type" : "string"
```

```
          },
          "source_host" : {
            "type" : "string"
          },
          "timestamp" : {
            "type" : "string"
          }
        }
      }
    }
  }
}
```

Podemos ver no JSON anterior que todos os campos que definimos estão devidamente tipados. Alguns campos, como `@timestamp` , já são padrão do indexador, por isso não precisamos defini-los.

## Fragmentos (shards)

Bom, então já entendemos o que são índices e o que são documentos e seus tipos, mas e o que seriam fragmentos (shards)? O leitor pode ter reparado quando vimos o console com a mensagem que indicava a criação do índice, além das informações retornadas nas consultas, menções a fragmentos, indicando que o índice foi criado com 5 fragmentos e que as consultas foram efetuadas nos 5 fragmentos.

Lembra de quando falamos que o Elasticsearch foi construído *sob* o Lucene? Pois bem, esses shards nada mais são do que índices do Apache Lucene que estão executando por debaixo dos panos! Concluindo nossa analogia, os shards seriam como partições de uma tabela em um banco de dados relacional, que possui como objetivo prover maior agilidade nas consultas.

Tipicamente, o Elasticsearch efetua um cálculo de hash em cima de cada documento no ato da indexação e o armazena em um dos fragmentos. Porém, como veremos adiante no livro, é possível

manipular o comportamento de armazenagem e busca nos shards a fim de maximizar a performance, embora o comportamento padrão do Elasticsearch costume ser suficiente.

Assim, concluímos o nosso entendimento. Ou seja, temos índices que armazenam documentos, que possuem seus mapeamentos estruturais, que por sua vez são distribuídos em fragmentos, que nada mais são do que índices do Apache Lucene.

## Cuidado com a tipagem dos campos de um documento!

Um último ponto muito importante sobre a tipagem de campos dentro de um documento é que os identificadores de campos são *cross-índice*, ou seja, um mesmo campo é criado para todo o índice, não apenas para o `document type`. Imaginemos um exemplo:

1. Criamos um `document type` com dados de cliente, entre os quais existe um campo chamado `telefone`, que criamos como um inteiro;

2. Agora, criamos um outro `document type` com dados de lojas físicas, que também possui um campo chamado `telefone`. Porém, nesse caso, o criamos como texto e introduzimos esse novo tipo de campo como texto, dentro do mesmo índice em que estavam as informações de cliente.

Ao realizar essa ação, geraremos um cenário de instabilidade, pois o Elasticsearch simplesmente passará a cadastrar todos os novos documentos – inclusive os do tipo cliente! – com o tipo texto no campo `telefone` que foi criado por último, gerando assim não conformidades. Por isso, deve-se ter a preocupação de sempre ter os campos muito bem definidos e separados, a fim de evitar surpresas desagradáveis no futuro.

### Padrões de URL da API do Elasticsearch e nomenclatura de índices

O leitor pode estar se perguntando o porquê de termos criado o nome do índice concatenando uma data ao final, que consequentemente nos obriga a utilizar o caractere `*` para nossas ações. A razão é muito simples: conforme veremos futuramente, é mais fácil realizar a expurga dos dados antigos deletando índices – pelo componente `curator` – do que removendo os documentos manualmente.

Com essa configuração, a cada novo dia que o nosso pipeline do Logstash estiver executando, será criado um novo índice com a data do dia como sufixo e utilizando o caractere `*`. Assim indicamos que desejamos que a ação que estamos realizando seja feita através de todos os índices existentes; no nosso caso, para todos os índices que têm como prefixo a palavra `casadocodigo`.

Por fim, já vimos nas nossas primeiras URLs chamando a API do Elasticsearch, e em todas as que usaremos, um padrão no formato da URL, com partes que variam de ação para ação, que consiste, a saber:

```
<ip>:<porta>/<índice>/<tipo do documento>/<ação>?<atributos>
```

## 3.4 AÇÕES DO ELASTICSEARCH

Tipicamente, temos dois tipos de ações no Elasticsearch:

- **Indexação:** nessa ação, os documentos são inseridos/alterados/excluídos. Um dado interessante é que, na verdade, os documentos não são de fato excluídos ou mesmo alterados, mas sim versionados e inativados, conforme veremos adiante no livro.

- **Busca:** nessa ação, a principal do Elasticsearch, efetuamos as buscas propriamente ditas, com a mais diversa gama de features, como identificação de língua, busca por sinônimos, agrupamentos e contagem (*max/min/avg*) de determinados eventos, expressões lógicas etc. Veremos neste capítulo e mais adiante diversos exemplos de uso desses recursos.

Assim, agora que esgotamos a nossa introdução ao funcionamento do Elasticsearch, vamos voltar à prática, populando o nosso indexador com uma massa de testes e fazendo umas consultas!

## 3.5 PREPARANDO A MASSA DE TESTES COM O APACHE JMETER

Para montarmos a nossa massa de testes, vamos utilizar uma ferramenta de testes muito conhecida, o Apache JMeter. O leitor pode encontra o link para download em http://jmeter.apache.org/download_jmeter.cg. Para instalar e usar o JMeter, é muito fácil: basta fazer o download do `zip`, descompactar e executar o seguinte comando, dentro da pasta `bin` do JMeter:

```
java -jar ApacheJMeter.jar
```

Após executar o comando, vamos nos deparar com a seguinte tela, feita em Java Swing:

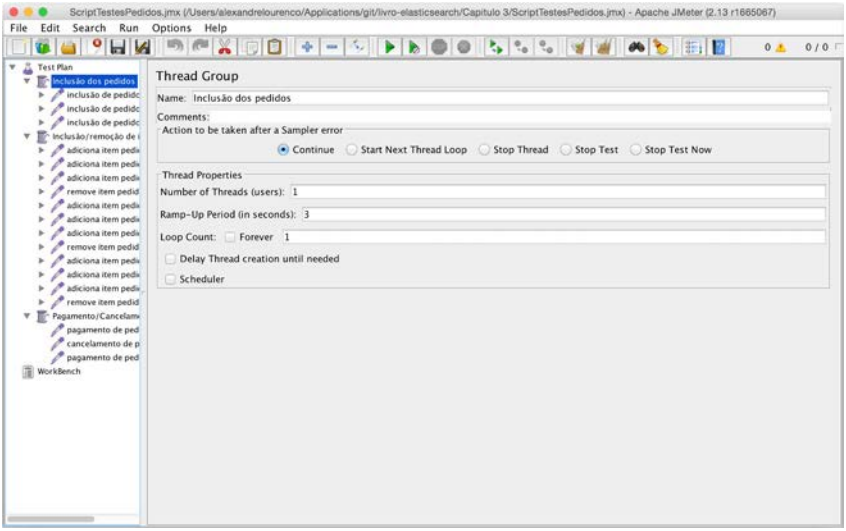



Figura 3.2: Apache JMeter

Eu já disponibilizei no meu repositório dois scripts de carga que efetuam a execução de diversas chamadas HTTPs às APIs, um script para a API de Clientes e outro para a API de Pedidos. Os scripts se encontram em https://github.com/alexandreesl/livro-elasticsearch/tree/master/Capitulo%203.

## Preparando a massa de clientes

Vamos então começar a montar a nossa massa de testes. Comecemos populando com dados de clientes, abrindo o script no arquivo `ScriptTestesClientes.jmx` . Após abrir o script no JMeter, basta executar o script com o botão `run` e pronto, temos uma massa de dados de clientes!

Você reparou em um ponto interessante na console da API? Algumas de nossas execuções apresentaram falha, graças ao nosso gerador de erros caótico! A seguir, podemos ver um trecho de log de uma execução de exemplo:

```
... omitido

21:14:12.073 [http-nio-8081-exec-10] INFO  br.com.alexandreesl.han
dson.rest.ClienteRestService - O cliente 6 foi alterado!
21:14:12.075 [http-nio-8081-exec-1] INFO  br.com.alexandreesl.hand
son.rest.ClienteRestService - O cliente 6 foi alterado!
21:14:12.077 [http-nio-8081-exec-2] INFO  br.com.alexandreesl.hand
son.rest.ClienteRestService - O cliente 7 foi alterado!
21:14:12.080 [http-nio-8081-exec-3] ERROR org.apache.catalina.core
.ContainerBase.[Tomcat].[localhost].[/].[jerseyServlet] - Servlet.
service() for servlet [jerseyServlet] in context with path [] thre
w exception [java.lang.RuntimeException: Ocorreu um erro caótico!]
 with root cause
java.lang.RuntimeException: Ocorreu um erro caótico!
    at br.com.alexandreesl.handson.rest.ClienteRestService.mergeCl
iente(ClienteRestService.java:113) ~[classes/:?]
    at sun.reflect.GeneratedMethodAccessor30.invoke(Unknown Source
) ~[?:?]
    at sun.reflect.DelegatingMethodAccessorImpl.invoke(DelegatingM
ethodAccessorImpl.java:43) ~[?:1.8.0_60]
    at java.lang.reflect.Method.invoke(Method.java:497) ~[?:1.8.0_
60]
    at org.glassfish.jersey.server.model.internal.ResourceMethodIn
vocationHandlerFactory$1.invoke(ResourceMethodInvocationHandlerFac
tory.java:81) ~[jersey-server-2.13.jar:?]
    at org.glassfish.jersey.server.model.internal.AbstractJavaReso
urceMethodDispatcher$1.run(AbstractJavaResourceMethodDispatcher.ja
va:151) ~[jersey-server-2.13.jar:?]
    at org.glassfish.jersey.server.model.internal.AbstractJavaReso
urceMethodDispatcher.invoke(AbstractJavaResourceMethodDispatcher.j
ava:171) ~[jersey-server-2.13.jar:?]
    at org.glassfish.jersey.server.model.internal.JavaResourceMeth
odDispatcherProvider$VoidOutInvoker.doDispatch(JavaResourceMethodD
ispatcherProvider.java:136) ~[jersey-server-2.13.jar:?]
    at org.glassfish.jersey.server.model.internal.AbstractJavaReso
urceMethodDispatcher.dispatch(AbstractJavaResourceMethodDispatcher
.java:104) ~[jersey-server-2.13.jar:?]
    at org.glassfish.jersey.server.model.ResourceMethodInvoker.inv
oke(ResourceMethodInvoker.java:384) ~[jersey-server-2.13.jar:?]
    at org.glassfish.jersey.server.model.ResourceMethodInvoker.app
ly(ResourceMethodInvoker.java:342) ~[jersey-server-2.13.jar:?]
    at org.glassfish.jersey.server.model.ResourceMethodInvoker.app
ly(ResourceMethodInvoker.java:101) ~[jersey-server-2.13.jar:?]
    at org.glassfish.jersey.server.ServerRuntime$1.run(ServerRunti
me.java:271) ~[jersey-server-2.13.jar:?]
    at org.glassfish.jersey.internal.Errors$1.call(Errors.java:271
) ~[jersey-common-2.13.jar:?]
    at org.glassfish.jersey.internal.Errors$1.call(Errors.java:267
```

```
) ~[jersey-common-2.13.jar:?]
    at org.glassfish.jersey.internal.Errors.process(Errors.java:315
) ~[jersey-common-2.13.jar:?]
    at org.glassfish.jersey.internal.Errors.process(Errors.java:297
) ~[jersey-common-2.13.jar:?]

... omitido
```

Isso é bom para os nossos testes, pois podemos verificar se nosso pipeline pode processar adequadamente também as mensagens de erro. Na figura seguinte do console, podemos ver como o *stacktrace* é montado dentro da mensagem:

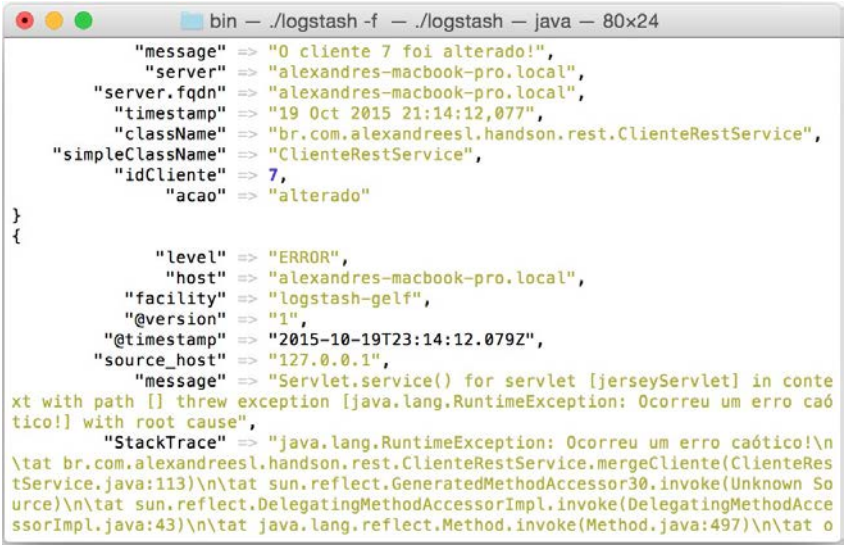



Figura 3.3: Console do logstash com trecho de stack trace

Agora, vamos verificar se de fato possuímos dados cadastrados no Elasticseach. Para isso, efetuaremos uma consulta, onde pesquisamos por todas as alterações de clientes:

```
curl -XGET 'localhost:9200/casadocodigo-*/_search?pretty&q=acao:alterado'
```

Essa consulta vai produzir uma consulta como a do trecho

adiante, onde temos listados todos os eventos de alterações de Clientes:


```
{
  "took" : 59,
  "timed_out" : false,
  "_shards" : {
    "total" : 10,
    "successful" : 10,
    "failed" : 0
  },
  "hits" : {
    "total" : 696,
    "max_score" : 2.738478,
    "hits" : [ {
      "_index" : "casadocodigo-2015.10.18",
      "_type" : "logs",
      "_id" : "AVB8hyeyYMia8wVICZc-",
      "_score" : 2.738478,
      "_source":{"level":"INFO","host":"alexandres-macbook-pro.loc
al","facility":"logstash-gelf","@version":"1","@timestamp":"2015-1
0-18T19:57:22.775Z","source_host":"127.0.0.1","message":"O cliente
 7 foi alterado!","server":"alexandres-macbook-pro.local","server.
fqdn":"alexandres-macbook-pro.local","timestamp":"18 Oct 2015 17:5
7:22,775","className":"br.com.alexandreesl.handson.rest.ClienteRes
tService","simpleClassName":"ClienteRestService","idCliente":7,"ac
ao":"alterado"}
    }, {
      "_index" : "casadocodigo-2015.10.18",
      "_type" : "logs",
      "_id" : "AVB8hyeyYMia8wVICZc_",
      "_score" : 2.738478,
      "_source":{"level":"INFO","host":"alexandres-macbook-pro.loc
al","facility":"logstash-gelf","@version":"1","@timestamp":"2015-1
0-18T19:57:22.778Z","source_host":"127.0.0.1","message":"O cliente
 7 foi alterado!","server":"alexandres-macbook-pro.local","server.
fqdn":"alexandres-macbook-pro.local","timestamp":"18 Oct 2015 17:5
7:22,778","className":"br.com.alexandreesl.handson.rest.ClienteRes
tService","simpleClassName":"ClienteRestService","idCliente":7,"ac
ao":"alterado"}
    },

    ... restante omitido
```


Agora, vamos testar popular mais alguns clientes e, dessa vez, não vamos excluí-los. Para isso, vamos alterar todos os JSONs para

que reflitam os IDs de novos clientes, como na figura a seguir. No último *thread group*, intitulado `Exclusão de clientes`, basta clicar com o botão direito e selecionar a opção `disable`:

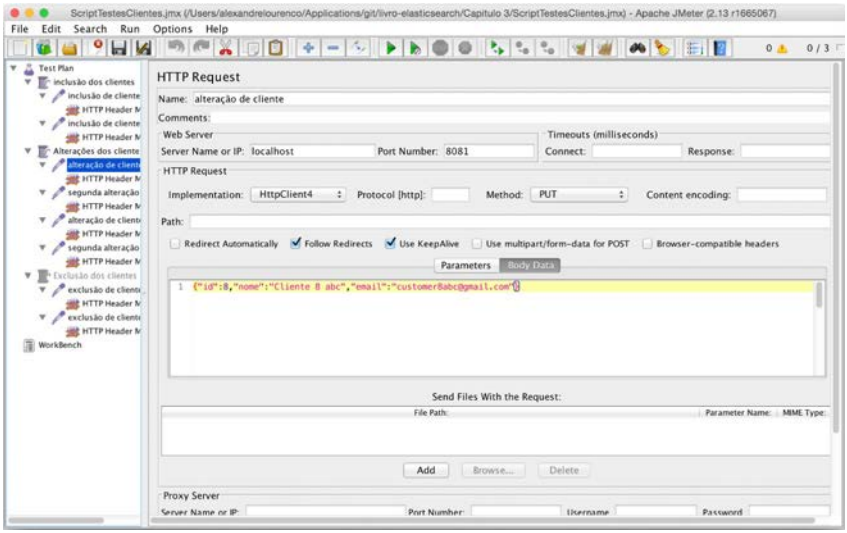



Figura 3.4: Alteração do JMeter

Agora, vamos fazer a mesma consulta que fizemos anteriormente. Você reparou que a quantidade de registros retornados não mudou? Isso acontece porque as consultas no Elasticsearch são paginadas, retornando por default os 10 documentos que obtiveram a maior pontuação no analisador do Elasticsearch – falaremos mais sobre esse assunto na seção *Analisadores e scores de documentos* –, partindo do mais relevante no score para o menos.

Para usar a paginação para retornar todos os resultados em páginas, basta utilizar os parâmetros `from` e `size`. Por exemplo, na consulta anterior, se quisermos que a consulta retorne os 25 primeiros resultados, podemos modificar da seguinte forma, por exemplo (o `from` neste caso poderia ser implícito, mas estamos usando apenas para demonstração):

```
curl -XGET 'localhost:9200/casadocodigo-*/_search?pretty&q=acao:al
terado&size=25&from=0'
```

O leitor poderá observar, ao contar novamente os registros, que agora a quantidade de registros retornada condiz com o esperado pelos parâmetros passados.

Agora que já temos nossa massa de clientes, vamos passar para a massa de pedidos.

## Preparando a massa de pedidos

Vamos agora passar para a massa de pedidos. Analogamente ao que tínhamos para a massa de clientes, vamos abrir o outro script que já temos preparado no arquivo `ScriptTestesPedidos.jmx` , e executá-lo como fizemos no caso anterior.

Após a execução, será possível identificar que, análogo ao nosso exemplo anterior, nosso gerador de erros caóticos voltou a atacar, simulando erros em nossa API. Agora, como fizemos anteriormente, vamos modificar o nosso script para incluirmos mais pedidos. Vamos também desabilitar o thread group *Pagamento/Cancelamento de pedidos*, de modo que os pedidos ficarão com o status de "em aberto". A figura seguinte ilustra as alterações:

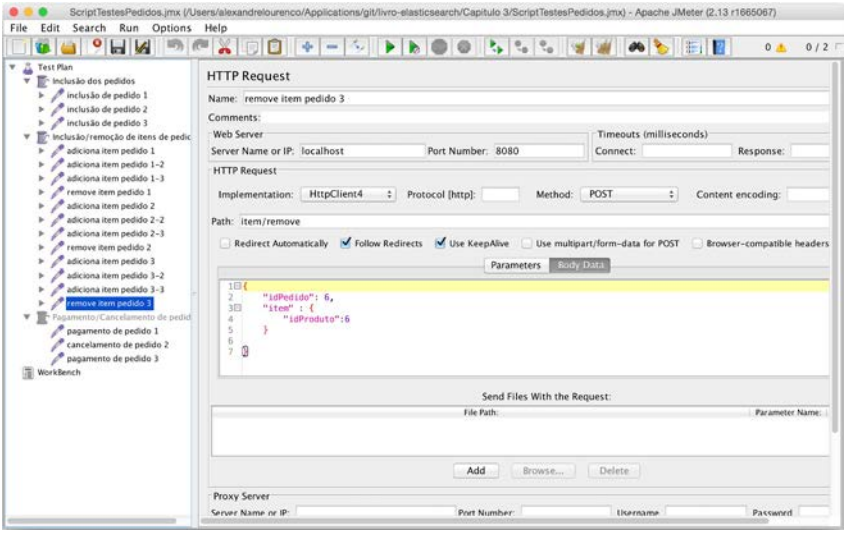



Figura 3.5: Alteração de pedido do JMeter

Após a execução, está concluído o preparo da nossa massa de testes. Analogamente ao que fizemos, vamos fazer uma consulta para testar a massa, buscando os registros de inclusões de itens, buscando a primeira página com 25 resultados por página:

```
curl -XGET 'localhost:9200/casadocodigo-*/_search?pretty&q=acaoIte
mPedido:adicionou&size=25&from=0'
```

Após executar o comando, você deve ter em seu console uma lista de eventos de logs de inclusões de pedidos, provando que a massa foi inserida com sucesso. Vamos testar agora algumas consultas com a nossa massa, porém, antes de prosseguirmos, vamos gastar um pouco de tempo para compreender mais a fundo como funcionam os mecanismos de busca do Elasticsearch.

## 3.6 ANALISADORES E SCORES DE DOCUMENTOS

Vamos parar para analisar as consultas que fizemos

anteriormente no Elasticsearch. Nela temos uma query, inserida dentro do parâmetro `q`, onde colocamos uma chave – uma campo do Elasticsearch que usaremos de filtro – e um valor, separado por um `:`. Você pode estar pensando que a avaliação da query se resume a um "se campo = valor, verdadeiro", entretanto, na prática, o que ocorre é que quando indexamos nossos eventos de log, o analisador default do Elasticsearch entra em ação, indexando as palavras do texto que compõe o campo e o armazenando.

Isso significa que, caso o valor fosse uma frase, nossa consulta retornaria *todos* os resultados cujos textos contidos naquele campo possuíssem a palavra, que na terminologia do Elasticsearch, chamamos de termo, que usamos como filtro. Vamos entender melhor agora como funcionam os analisadores.

## Analisadores

Analisadores no Elasticsearch, como o próprio nome diz, são responsáveis por analisar os conteúdos textuais que estamos fornecendo para a indexação. Esses conteúdos são analisados pelos diferentes componentes que compõem um analisador, que são:

- **Character filters:** esses filtros são responsáveis por eliminar todos os caracteres não textuais antes da execução do *tokenizer*, a fim de que esses caracteres não atrapalhem a indexação. Um exemplo de ação desses filtros é a remoção de código HTML no meio de um texto.

- **Tokenizer:** muito comum em diversas linguagens de programação, um tokenizer consiste em um componente responsável por efetuar a decomposição de um texto em diversos tokens, comumente palavras, a fim permitir a avaliação de cada termo encontrado no

texto. Após a execução do tokenizer, podemos ter 0 ou mais filtros, dos quais falaremos a seguir.

- **TokenFilters:** esses filtros são responsáveis por efetuar os mais diversos tipos de análises ou transformações nos dados para permitir uma melhor busca, como transformar o texto para *lower case*, com o objetivo de eliminar distorções de case nas consultas, análise de regras de sinônimos para permitirmos buscas baseadas em sinônimos, e até mesmo análises da língua do texto. Tudo isso não só com o objetivo de identificar a língua ao qual o texto pertence, como também para permitir que seja encontrado o termo "base" que compõe cada token. Todos esses filtros permitem um mecanismo de consulta muito rico, permitindo, por exemplo, que para uma dada palavra "aventura" também sejam retornados resultados para as palavras "aventureiro", "aventureira", "aventurados" etc.

Esses `tokenfilters` são executados sequencialmente um após o outro, ou seja, os resultados de um geram o input para a execução do outro. É possível modificar tanto o analisador que será usado na indexação quanto usar um analisador diferente apenas nas buscas.

Para o nosso exemplo atual, os analisadores default já satisfazem as nossas necessidades, mas conheceremos melhor os tipos de analisadores disponíveis e veremos alguns de seus usos na prática mais adiante no livro. Outro ponto importante a se notar é que, como já falamos antes, nosso indexador é baseado no Lucene, portanto, os analisadores que utilizamos no Elasticsearch são mapeados para equivalentes no Lucene.

No ato da busca, a partir da análise feita pelo analisador, o Lucene retorna um score, que indica o quanto aquele determinado

documento é aderente – ou em outras palavras, relevante – aos critérios de busca fornecidos. Vamos agora entender um pouco sobre o funcionamento desse score.

## Score

Quando o Elasticsearch nos retorna os dados de uma consulta, esses dados são baseados no score, ou relevância, onde são retornados os documentos da ordem do mais relevante para o menos relevante. Vamos sempre ter em mente que, sempre que falamos de documentos em Elasticseacrh, estamos nos referindo aos registros de log que cadastramos de nossas APIs na prática que estamos desenvolvendo.

O nome do algoritmo usado pelo Lucene é o *TF/IDF*, que no original, em inglês, significa *term frequency/inverse document frequency* (frequência de termo/frequência inversa de documento, em uma tradução livre).

Para entender melhor o significado desse algoritmo, não é necessário conhecer a sua fórmula completa, bastando compreender os 3 principais conceitos aplicados nele, a saber:

- **Frequência do termo:** como o próprio nome diz, este critério avalia a quantidade de incidências do termo dentro do campo pesquisado em cada documento.
- **Frequência inversa de documento:** neste critério, temos basicamente o inverso do critério anterior, onde quanto maior a incidência do termo em todos os documentos do índice, menos relevantes os resultados serão. O objetivo desse critério é auxiliar na filtragem de termos muito abrangentes (comuns) de termos mais incomuns.

- **Comprimento do campo:** esse critério mede o tamanho do campo onde o termo foi encontrado, considerando que quanto maior for o comprimento do campo, menor a relevância daquele documento na consulta. O racional desse critério baseia-se no fato de que textos longos tendem a ser mais abrangentes, logo a acurácia do resultado não é a mesma do que em textos menores.

É bom frisar que, além desses termos, outros são aplicados, como a proximidade do termo com o texto exato informado na consulta, por exemplo. Em consultas muito complexas, internamente o indexador pode executar múltiplas subqueries para formar o resultado, combinando os scores das subqueries para formar o score final dos resultados.

Caso você esteja desenvolvendo uma query complexa no Elasticsearch, é interessante conhecer o parâmetro `explain`, que fornece uma forma de depurar uma query, dando os detalhes não só da query principal da busca como também das subqueries usadas para o processamento dos resultados, se aplicável.

Vamos ver um exemplo de uso desse parâmetro na prática. Vamos executar uma das consultas de pedidos que realizamos anteriormente, executando-a com o parâmetro `explain` e o comando a seguir:

```
curl -XGET 'localhost:9200/casadocodigo-*/_search?pretty&q=acaoIte
mPedido:adicionou&explain'
```

Após realizarmos a consulta, podemos ver que, para cada registro do resultado, existe uma estrutura JSON de título `_explanation`, onde os detalhes relativos ao cálculo da relevância são apresentados. Conforme dito anteriormente, esse output pode ser muito útil em cenários complexos, em que podemos analisar de

que forma a relevância está sendo processada. No fragmento a seguir, podemos ver um exemplo desse retorno:


```
{
 "took" : 6,
 "timed_out" : false,
 "_shards" : {
   "total" : 5,
   "successful" : 5,
   "failed" : 0
 },
 "hits" : {
   "total" : 1250,
   "max_score" : 1.7514161,
   "hits" : [ {
     "_shard" : 0,
     "_node" : "__3RbNMwSLya2hcl4ESCRA",
     "_index" : "casadocodigo-2015.10.27",
     "_type" : "logs",
     "_id" : "AVCmuc04MYVF93THYgju",
     "_score" : 1.7514161,
     "_source":{"level":"INFO","host":"alexandres-macbook-pro.loc
al","facility":"logstash-gelf","@version":"1","@timestamp":"2015-1
0-27T00:36:46.000Z","source_host":"127.0.0.1","message":"pedido 1
do cliente 1 adicionou o produto 2","server":"alexandres-macbook-p
ro.local","server.fqdn":"alexandres-macbook-pro.local","timestamp":
"26 Oct 2015 22:36:46,000","className":"br.com.alexandreesl.handso
n.rest.PedidoRestService","simpleClassName":"PedidoRestService","i
dPedido":1,"idCliente":1,"acaoItemPedido":"adicionou","idProdutoPe
dido":2},
     "_explanation" : {
       "value" : 1.7514161,
       "description" : "weight(acaoItemPedido:adicionou in 58) [P
erFieldSimilarity], result of:",
       "details" : [ {
         "value" : 1.7514161,
         "description" : "fieldWeight in 58, product of:",
         "details" : [ {
           "value" : 1.0,
           "description" : "tf(freq=1.0), with freq of:",
           "details" : [ {
             "value" : 1.0,
             "description" : "termFreq=1.0"
           } ]
         }, {
           "value" : 1.7514161,
           "description" : "idf(docFreq=224, maxDocs=477)"
```

```
          }, {
            "value" : 1.0,
            "description" : "fieldNorm(doc=58)"
          } ]
        } ]
      }
    }
```

Agora que compreendemos como funcionam os indexadores e o score, vamos seguir adiante nos nossos estudos, experimentando realizar algumas consultas em nosso cluster!

## 3.7 CONSULTAS BÁSICAS DO ELASTICSEARCH

### Consultas por termos

A consulta por termos é uma das mais simples formas de pesquisa do Elasticsearch, nada mais sendo do que o tipo de query que temos realizado no decorrer deste capítulo. Nesse tipo de consulta, é consultado no indexador por documentos cujo texto contenha o termo passado.

É importante ter em mente que, nesse modelo de query, a consulta não utiliza recursos de análise que citamos anteriormente, como sinônimos etc., mas apenas checando pela incidência exata do termo passado e aplicando o cálculo de score que vimos anteriormente para calcular a relevância dos resultados.

Além do formato de `query parameter` , também é possível realizar as consultas enviando as filtragens como uma estrutura JSON, como no exemplo:

```
curl -XGET 'http://localhost:9200/casadocodigo-*/_search?pretty' -d
 '{
  "query": {
    "term": {
      "acaoItemPedido": "adicionou"
    }
  }
```

```
}'
```

## Consultas por múltiplos termos

Assim como uma consulta pode ser feita sob um termo, podemos também ter uma consulta sob múltiplos termos. Para isso, basta realizarmos uma query do tipo `terms` – atenção ao **s**! – em vez de realizarmos a pesquisa da forma que temos feito até o momento. Na pesquisa de termos, temos uma consulta de lógica `OR`, ou seja, que vai retornar todos os documentos que contêm no campo pesquisado o *termo1* OU o *termo2* OU o *termo3*, e assim por diante.

Por exemplo, para pesquisarmos todos os documentos que contenham no campo `StackTrace` os termos "ocorreu" ou "erro", basta realizarmos uma consulta como a seguinte:

```
curl -XGET 'http://localhost:9200/casadocodigo-*/_search?pretty' -d
 '{
  "query": {
    "terms": {
      "StackTrace": ["ocorreu","erro"]
    }
  }
}'
```

## Consultas por múltiplos campos (query_string)

Muitas vezes, temos a necessidade de efetuar uma consulta por múltiplos campos, envolvendo também múltiplas operações lógicas, como em uma consulta tradicional em um banco de dados relacional, por exemplo.

Vamos supor que desejamos consultar todos as adições *OU* remoções de itens em pedidos *E* que sejam do produto de id `6`. Como faríamos essa consulta? Por meio do recurso `query_string` do Elasticsearch. Basta informarmos:

```
curl -XGET 'http://localhost:9200/casadocodigo-*/_search?pretty' -d
 '{
  "query": {
    "query_string" : {
        "query": "(acaoItemPedido:adicionou OR acaoItemPedido:remo
veu) AND idProdutoPedido:6"
    }
  }
}'
```

Ao executarmos essa consulta, veremos no nosso console os resultados de nossa query, provando que a nossa consulta foi um sucesso:

```
{
  "took" : 6,
  "timed_out" : false,
  "_shards" : {
    "total" : 5,
    "successful" : 5,
    "failed" : 0
  },
  "hits" : {
    "total" : 892,
    "max_score" : 1.40223,
    "hits" : [ {
      "_index" : "casadocodigo-2015.10.27",
      "_type" : "logs",
      "_id" : "AVCmxUfOMYVF93THYgzT",
      "_score" : 1.40223,
      "_source":{"level":"INFO","host":"alexandres-macbook-pro.loc
al","facility":"logstash-gelf","@version":"1","@timestamp":"2015-1
0-27T00:49:17.161Z","source_host":"127.0.0.1","message":"pedido 13
 do cliente 4 removeu o produto 6","server":"alexandres-macbook-pr
o.local","server.fqdn":"alexandres-macbook-pro.local","timestamp":
"26 Oct 2015 22:49:17,162","className":"br.com.alexandreesl.handso
n.rest.PedidoRestService","simpleClassName":"PedidoRestService","i
dPedido":13,"idCliente":4,"acaoItemPedido":"removeu","idProdutoPed
ido":6}
    }

... restante omitido
```

## Consultas por prefixo

Outro tipo de consulta é a por prefixo que, como o próprio nome diz, realiza consultas buscando pelo prefixo, isto é, de documentos cujo *início* do texto começa com o termo passado na consulta.

Por exemplo, se quiséssemos consultar todos os documentos cujo campo `acao` comece com o texto `alt`, basta realizarmos uma consulta como a seguinte:

```
curl -XGET 'http://localhost:9200/casadocodigo-*/_search?pretty' -d
 '{
  "query": {
     "prefix" : {
         "acao" : "alt"
         }
  }
}'
```

## Consultas utilizando expressões regulares

Outra forma de realizarmos consultas é pelas expressões regulares, na qual a consulta aplica padrões (expressões) nas buscas ao contrário de termos literais, permitindo inclusive a utilização de *wildcards*.

Por exemplo, vamos supor que quiséssemos fazer a mesma consulta que fizemos anteriormente, porém, em vez de pesquisarmos por prefixo, pesquisaríamos pela incidência da sequência `alt` seguido de 5 caracteres, excluindo assim qualquer coisa que fosse diferente da palavra 'alterado'. Poderíamos realizar essa proposta de consulta da seguinte forma:

```
curl -XGET 'http://localhost:9200/casadocodigo-*/_search?pretty' -d
 '{
  "query": {
     "regexp":{
        "acao": "alt....."
    }
  }
}'
```

Para saber mais sobre as possibilidades de utilização de expressões regulares no Elasticsearch, sugiro consultar sua documentação relacionada a esse assunto, em https://www.elastic.co/guide/en/elasticsearch/reference/current/query-dsl-regexp-query.html

## Consultas utilizando wildcards

Também é possível utilizar wildcards nas consultas de termos. Eles são caracteres reservados comumente usados em tecnologia, como quando queremos efetuar uma pesquisa no nosso `file explorer` favorito e queremos pesquisar em todos os arquivos do tipo `txt`, usamos o caractere '*, no caso,* '.txt'.

Por exemplo, vamos supor, seguindo a mesma consulta dos nossos dois exemplos anteriores, que quiséssemos pesquisar por todos os eventos onde no campo ação tivéssemos um valor de prefixo `alt`, seguido de qualquer sequência de caracteres, e finalizando com o caractere `o`. Assim ficaria a nossa query, utilizando a query de wildcards:

```
curl -XGET 'http://localhost:9200/casadocodigo-*/_search?pretty' -d
 '{
  "query": {
    "wildcard" : { "acao" : "alt*o" }
  }
}'
```

## Consultas utilizando lógica Fuzzy

De acordo com o Wikipedia (link: https://pt.wikipedia.org/wiki/Lógica_difusa), a lógica fuzzy consiste de:

A lógica difusa (ou lógica fuzzy) é uma extensão da lógica booleana que admite valores lógicos intermediários entre o FALSO (0) e o VERDADEIRO (1); por exemplo, o valor médio 'TALVEZ' (0,5). Isto significa que um valor lógico difuso é um valor qualquer no intervalo de valores entre 0 e 1. Este tipo de lógica engloba de certa forma conceitos estatísticos principalmente na área de Inferência.

Ou seja, chamamos de lógica fuzzy um conceito lógico onde é considerado não apenas os valores verdadeiro e falso, mas também *nuances* que podem existir entre esses valores.

Dentro do conceito do Elasticsearch, podemos utilizar queries de lógica fuzzy tanto para campos texto quanto para campos numéricos e temporais. Para campos texto, a query se utiliza de um algoritmo chamado de distância de Levenshtein, que calcula a quantidade de operações necessárias para que, a partir de um termo base, se possa chegar a outro termo, onde por operação entendemos a inclusão, exclusão ou alteração de um caractere.

Por exemplo, para o termo `casa`, temos uma distância de Levenshtein de 3 do termo `casarao`, pois são 3 operações de adição de caracteres para se transformar uma palavra em outra.

Para consultar, por exemplo, no campo `message` – que contém o texto *bruto* dos nossos eventos de log – por todos os documentos a partir do termo `ped` usando de lógica fuzzy, basta efetuarmos uma consulta como a seguir:

```
curl -XGET 'http://localhost:9200/casadocodigo-*/_search?pretty' -d
 '{
  "query": {
    "fuzzy" : { "message" : "ped" }
  }
```

```
}'
```

Quando executarmos essa query, caso o leitor tenha utilizado o pipeline do logstash para logar também o log de inicialização do Spring Boot, vai notar que alguns eventos de log como o do trecho seguinte também são retornados pela pesquisa:

```
... restante omitido
{
      "_index" : "casadocodigo-2015.10.18",
      "_type" : "logs",
      "_id" : "AVB8hnRIYMia8wVICZbv",
      "_score" : 1.18678,
      "_source":{"level":"INFO","host":"alexandres-macbook-pro.loc
al","facility":"logstash-gelf","@version":"1","@timestamp":"2015-1
0-18T19:56:37.815Z","source_host":"127.0.0.1","message":"Starting
Application on Alexandres-MacBook-Pro.local with PID 2748 (started
 by alexandrelourenco in /Users/alexandrelourenco/Applications/git
/livro-Elasticsearch/Capitulo 1/Cliente-REST)","server":"alexandre
s-macbook-pro.local","server.fqdn":"alexandres-macbook-pro.local",
"timestamp":"18 Oct 2015 17:56:37,816","className":"org.springfram
ework.boot.StartupInfoLogger","simpleClassName":"StartupInfoLogger'
,"logdefault":"Starting Application on Alexandres-MacBook-Pro.loca
l with PID 2748 (started by alexandrelourenco in /Users/alexandrel
ourenco/Applications/git/livro-Elasticsearch/Capitulo 1/Cliente-RE
ST)"}
    }
... restante omitido
```

A razão para isso é muito simples: na mensagem de log, temos o termo `PID`, que possui uma distância de Levenshtein de 1 em relação ao termo `ped`, que consultamos. Portanto, ele entrou nos nossos resultados devido a essa análise.

No caso dos campos numéricos, a consulta é feita por *range*, onde definimos um range tanto para mais quanto para menos, com relação a um número base que fornecemos como parâmetro de consulta. Por exemplo, para a query:

```
curl -XGET 'http://localhost:9200/casadocodigo-*/_search?pretty' -d
 '{
  "query": {
   "fuzzy" : {
        "idProdutoPedido" : {
```

```
            "value" : 62,
            "fuzziness" : 5
        }
    }
  }
}'
```

Temos uma consulta onde obtemos todos os eventos de inclusão/remoção de itens em pedidos cujo id de produto vai de 57 (62-5) até 67 (62+5), cálculo este feito a partir do parâmetro `fuziness`. Se executarmos a query, poderemos ver pelo trecho a seguir que a consulta foi feita como o esperado:

```
...restante omitido
, {
      "_index" : "casadocodigo-2015.10.27",
      "_type" : "logs",
      "_id" : "AVCmw1G-MYVF93THYgt_",
      "_score" : 1.0,
      "_source":{"level":"INFO","host":"alexandres-macbook-pro.loc
al","facility":"logstash-gelf","@version":"1","@timestamp":"2015-1
0-27T00:47:08.259Z","source_host":"127.0.0.1","message":"pedido 1
do cliente 1 adicionou o produto 67","server":"alexandres-macbook-
pro.local","server.fqdn":"alexandres-macbook-pro.local","timestamp'
:"26 Oct 2015 22:47:08,259","className":"br.com.alexandreesl.hands
on.rest.PedidoRestService","simpleClassName":"PedidoRestService","
idPedido":1,"idCliente":1,"acaoItemPedido":"adicionou","idProdutoP
edido":67}
    } ]
  }
}
```

Para consultas por data (temporais), temos o mesmo conceito aplicado, por exemplo, para pesquisas por range de dias.

## Combinando múltiplas queries (filtered query)

Em todos os nossos exemplos anteriores, usamos uma única query. Dentro do Elasticseach, temos o conceito de *filtered queries*, que nada mais são do que filtros onde 'embutimos' queries de qualquer espécie. Essa estratégia traz duas grandes vantagens:

- A filtered query não precisará executar os cálculos de

score, bastando para ela saber quais documentos satisfazem a condição utilizada dentro do filtro;

- Filtered queries podem ter os seus filtros cacheados quando utilizados em conjunto, obtendo um grande incremento de performance.

Mais adiante no capítulo "Elasticsearch avançado", veremos mais detalhes sobre as filtered queries e seu mecanismo de cacheamento.

## 3.8 PLUGINS

Puxa vida, quanta coisa temos no Elasticsearch, certo? De fato, ele apresenta um set bem impressionante de funcionalidades disponibilizadas *out-of-the-box*. Contudo, dentro do Elasticsearch, também temos o conceito de plugins, que nada mais são do que extensões para o nosso indexador. Vamos testar um desses plugins agora, o kopf.

Com o kopf, temos uma interface gráfica administrativa para o nosso cluster, onde podemos ver informações relacionadas à saúde dele, além de uma interface para executarmos nossas queries REST das APIs do Elasticsearch.

Para instalar o kopf, primeiro vamos parar a nossa instância do Elasticsearch. Com a instância parada, vamos executar o seguinte comando:

```
bin/plugin install lmenezes/Elasticsearch-kopf
```

Dando tudo certo, veremos uma mensagem como esta no console:

```
-> Installing lmenezes/Elasticsearch-kopf...
Trying https://github.com/lmenezes/Elasticsearch-kopf/archive/mast
er.zip...
```

```
Downloading ..................................................
DONE
Installed kopf into /Users/alexandrelourenco/Applications/elk/elas
ticsearch-2.1.0/plugins/kopf
```

Após a instalação, vamos subir novamente o Elasticsearch. Para acessar o kopf, vamos abrir a seguinte URL no navegador:

```
http://localhost:9200/_plugin/kopf
```

O browser abrirá uma tela como a seguinte:

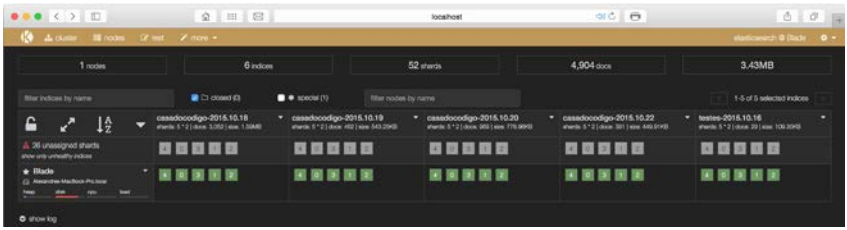



Figura 3.6: Home do kopf

Nessa tela podemos ver várias informações relacionadas ao cluster, como os índices criados, os shards, o status de cada shard, além de dados relacionados ao consumo de storage, CPU, heap etc. Para utilizar a interface de teste das APIs do Elasticsearch, basta clicar no ícone REST da interface, que nos leva a uma interface muito semelhante à do Postman:

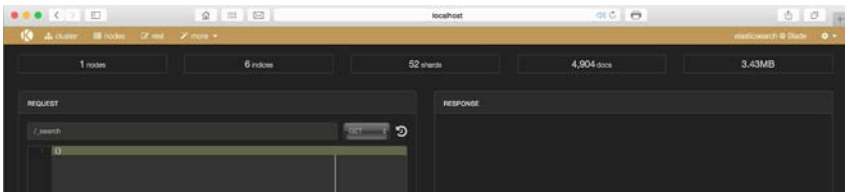



Figura 3.7: Tela de teste de APIs do kopf

Mais adiante no livro, veremos outros exemplos de plugins, como o shield, por exemplo, utilizado para implementar uma camada de segurança no nosso cluster.

## 3.9 CONCLUSÃO

E assim concluímos o nosso primeiro capítulo sobre o Elasticsearch. Espero que tenha conseguido passar para você, leitor, uma boa ideia do poder que temos em nossas mãos, ainda que não tenhamos resolvido ainda o problema das dashboards para a diretoria. Vamos agora para o próximo capítulo, onde finalmente vamos resolver a demanda do nosso chefe!

Capítulo 4

# DISSECANDO A ELK – KIBANA

## 4.1 DESENVOLVENDO RICAS INTERFACES PARA OS NOSSOS DADOS DE LOG

Figura 4.1: Logo do Kibana

Nos capítulos anteriores, vimos como criar pipelines de informações oriundas de nossas APIs com o Logstash, e como utilizar as informações desses pipelines com os poderosos mecanismos de busca do Elasticsearch. Agora, vamos dar uma cara *cool* para as nossas informações, criando as dashboards que nos foram pedidas!

## 4.2 CONHECENDO O KIBANA

O Kibana foi desenvolvido pela Elastic com o intuito de fornecer uma interface rica que permita consultas analíticas e/ou a construção de dashboards, com base nas informações contidas dentro de um cluster Elasticsearch.

A aplicação é toda construída em HTML e JavaScript e, a partir da versão 4.0, passou a ser disponibilizada com um Node.js embutido, em contrapartida com as versões anteriores, onde tínhamos de instalar um servidor web e implantar o Kibana dentro desse servidor.

A base de dados do Kibana é o próprio Elasticsearch, encapsulada dentro do índice `.kibana`. Dentro desse índice, estão contidas informações relacionadas aos metadados que o Kibana necessita para operar, como os índices que este está configurado para minerar, configurações de dashboards etc.

## 4.3 INSTALAÇÃO DO KIBANA

Vamos começar nosso estudo instalando o Kibana. Sua instalação é análoga às das outras ferramentas da nossa stack: basta entrar em https://www.elastic.co/downloads/kibana e realizar o download do arquivo. Após baixar e descompactar o arquivo, vamos até a instalação do Kibana, entramos na pasta `config` e abrimos o arquivo `kibana.yml` em um editor.

Dentro do editor, checamos a propriedade `elasticsearch_url`, que por default aponta para o host local (`localhost`) e a porta `9200`. Como estamos utilizando a configuração default do Elasticsearch, não precisamos mexer nessa configuração, mas é bom você saber onde será preciso mexer caso você mude a porta default.

Para testar a aplicação, vamos abrir uma nova janela de terminal, navegar para a pasta de instalação do Kibana – com o Elasticsearch rodando! – e executar o seguinte comando:

```
./bin/kibana
```

Após alguns instantes, podemos ver que o Kibana foi

inicializado com sucesso no nosso ambiente, como podemos ver no trecho de log a seguir:

```
....restante omitido....
[13:58:21.447] [info][status][plugin:kibana] Status changed from u
ninitialized to green - Ready
[13:58:21.476] [info][status][plugin:elasticsearch] Status changed
 from uninitialized to yellow - Waiting for Elasticsearch
[13:58:21.486] [info][status][plugin:kbn_vislib_vis_types] Status
changed from uninitialized to green - Ready
[13:58:21.489] [info][status][plugin:markdown_vis] Status changed
from uninitialized to green - Ready
[13:58:21.492] [info][status][plugin:metric_vis] Status changed fr
om uninitialized to green - Ready
[13:58:21.494] [info][status][plugin:spyModes] Status changed from
 uninitialized to green - Ready
[13:58:21.496] [info][status][plugin:statusPage] Status changed fr
om uninitialized to green - Ready
[13:58:21.498] [info][status][plugin:table_vis] Status changed fro
m uninitialized to green - Ready
[13:58:21.552] [info][listening] Server running at http://0.0.0.0:
5601
[13:58:21.706] [info][status][plugin:elasticsearch] Status changed
 from yellow to green - Kibana index ready
```

Para testar a instalação, vamos abrir um browser e entrar no endereço:

```
http://localhost:5601/
```

Após entrar no endereço, vamos ver uma tela inicial do Kibana quando ele é aberto pela primeira vez, solicitando a configuração de um índice. Nas próximas seções, vamos ver como configuraremos o Kibana para a nossa utilização. Antes de darmos início a essa parte, porém, vamos voltar ao arquivo de configuração e ver o que mais podemos fazer com ele!

## 4.4 CONFIGURANDO O KIBANA

### Configurações administrativas

Conforme vimos há pouco, todas a configurações que podemos fazer no Kibana se encontram no arquivo `kibana.yml` , dentro da pasta `config` . Segue um apanhado das configurações que podemos realizar nesse arquivo:

- `port` : configura a porta onde o Kibana responderá as requisições.
- `host` : configura o ip onde o Kibana vai responder as requisições.
- `elasticsearch_url` : conforme vimos anteriormente, é nesta configuração que definimos o local do Elasticsearch que vamos usar com o nosso Kibana.
- `elasticsearch_preserve_host` : configura o host utilizado pelo Kibana para identificar o solicitante das requisições para o Elasticsearch. Se for igual a `true` , utiliza o valor do host do solicitante (browser) da requisição para o Kibana; e se for igual a `false` , utiliza o host do Elasticsearch definido na propriedade anterior.
- `kibana_index` : nome do índice onde o Kibana efetua a gravação de seus metadados, como falamos anteriormente. O default é `.kibana` .
- `kibana_elasticsearch_username` : se o Elasticsearch estiver com autenticação habilitada – através do plugin shield, que veremos no capítulo *Administrando um cluster Elasticsearch* – , nesta configuração é definido o usuário para autenticação do Kibana.
- `kibana_elasticsearch_password` : se o Elasticsearch estiver com autenticação habilitada, nesta configuração

é definida a senha do usuário para autenticação do Kibana.

- `kibana_elasticsearch_client_crt` : se o Elasticsearch estiver configurado com segurança de certificação digital, neste campo é definido o caminho para o certificado digital que o Kibana utilizará para sua conexão.

- `kibana_elasticsearch_client_key` : se o Elasticsearch estiver configurado com segurança de certificação digital, neste campo é definido o caminho para a chave de certificado digital que o Kibana usará para sua conexão.

- `ca` : se o Elasticsearch estiver configurado com segurança de certificação digital, neste campo é definido o caminho para o cerificado CA que o Kibana utilizará para sua conexão.

- `default_app_id` : dentro do Kibana, cada seção da aplicação é considerada como uma aplicação independente. Neste campo definimos qual aplicação deve ser carregada por default, no caso a `discover` , que permite análises analíticas *drill-down* dos dados, como veremos a seguir.

- `ping_timeout` : tempo em milissegundos que o Kibana esperará pelo retorno do Elasticsearch em uma requisição de ping.

- `request_timeout` : tempo em milissegundos que o Kibana esperará pelo retorno do Elasticsearch nas requisições de consulta.

- `shard_timeout` : tempo em milissegundos que o Kibana esperará pelo retorno do Elasticsearch nas requisições de consulta utilizando-se de shards (fragmentos). Por default, este `timeout` está desabilitado.

- `startup_timeout` : tempo em milissegundos que o Kibana esperará pelo retorno do Elasticsearch durante a sua inicialização, caso ele não se apresente disponível.

- `verify_ssl` : flag indicando se o Kibana deve validar a autenticidade das conexões SSL (HTTPS) dos usuários do Kibana.

- `ssl_key_file` : configura a chave do certificado SSL caso configuremos o Kibana para utilizar conexões SSL de duas vias (em formato PEM).

- `ssl_cert_file` : configura o certificado SSL caso configuremos o Kibana para utilizar conexões SSL de duas vias (em formato PEM).

- `pid_file` : configura o caminho onde o Kibana criará o arquivo PID, permitindo que ele seja configurado para executar como um serviço do sistema operacional.

- `log_file` : configura o local onde o arquivo de log do Kibana deve ser criado. Ao configurar esta propriedade, o log da console do Kibana é desativado.

Agora que temos as configurações que podemos realizar com o Kibana devidamente configuradas, vamos começar a nossa prática.

## 4.5 EXECUTANDO O KIBANA PELA PRIMEIRA

VEZ

Para este capítulo, vamos utilizar o nosso Elasticsearch já populado anteriormente. Assim sendo, começaremos a usar o Kibana.

Vamos abrir um browser, digitar o endereço `http://localhost:5601/` e aguardar até que a tela seja renderizada. Vamos nos deparar com uma tela que aparece sempre que instalamos o Kibana pela primeira vez. Nessa tela, podemos ver que ele está solicitando que configuremos um índice do Elasticsearch para ele trabalhar, além de uma mensagem falando sobre uma falha para obtenção (*fetch*) de um mapeamento. A razão para isso é porque modificamos o valor default que o Logstash usa para a criação dos índices, com o prefixo `logstash` . A figura a seguir, extraída da tela, mostra essa situação para nós:

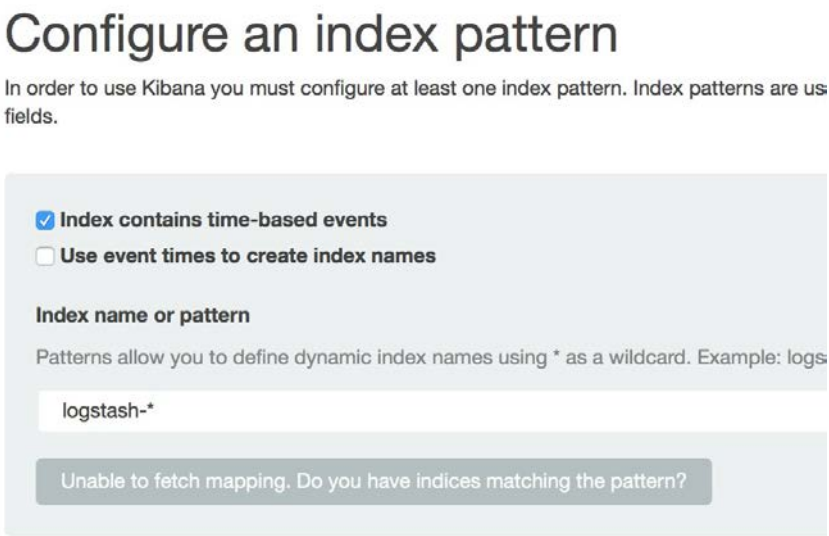



Figura 4.2: Tela inicial da primeira execução do Kibana

Dando prosseguimento, vamos fazer as devidas configurações e avançar na nossa construção. Para isso, no campo *Index name or pattern*, vamos modificar para o valor `casadocodigo-*` e pressionar `Tab`. Podemos ver agora que a configuração está nos pedindo que indiquemos o campo de filtragem de tempo (timestamp) do índice que vamos configurar.

Em todas as visões do Kibana, sempre temos uma filtragem mestra por período de tempo para o qual estamos consultando as informações, sendo nesse campo que definimos a forma como essa filtragem é feita. No nosso caso, usaremos o campo `@timestamp` já existente, que atende as nossas necessidades. Podemos ver a tela já configurada na figura:

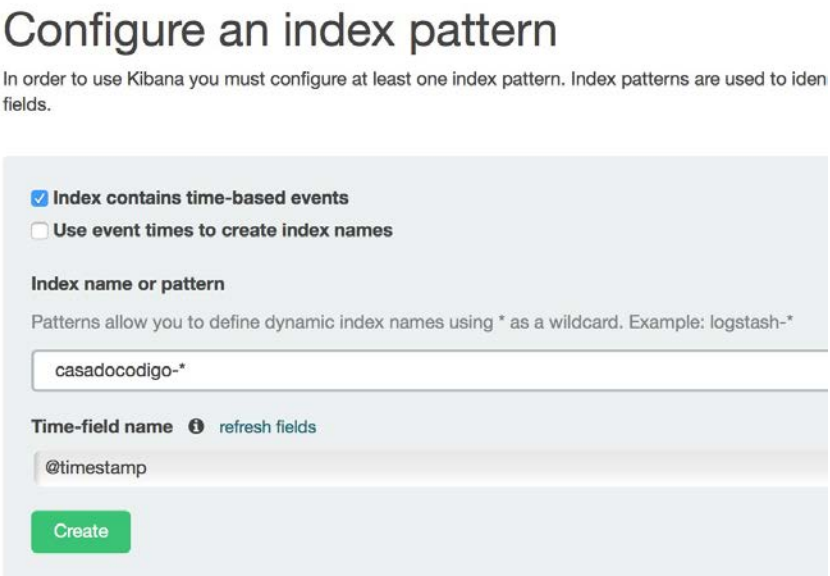



Figura 4.3: Tela inicial do Kibana configurada

Vamos clicar no botão *Create*. Seremos transportados para uma tela onde podemos ver o nosso índice configurado, inclusive com os campos mapeados devidamente para os tipos necessários,

respeitando as configurações que havíamos feito pelo plugin `mutate`.

A tela também contém 3 botões: um para definir que este índice deve ser o default a ser carregado (botão verde); outro que realiza o refresh dos campos de mapeamento, caso tenhamos feito modificações no pipeline (botão amarelo); e um que deleta as configurações desse índice no Kibana (botão vermelho).

Figura 4.4: Tela de configurações de índice

Vamos explorar um pouco mais a seção de configurações do Kibana. Na seção *Advanced*, podemos encontrar diversas configurações avançadas da ferramenta, como modificar a precisão dos campos numéricos, os tipos de máscaras de data aceitos, o formato em que os campos numéricos devem ser exibidos etc.

Estas configurações devem ser modificadas com cuidado, como o próprio aviso na página informa. Na próxima tela de configuração, podemos ver uma tela de edição de objetos salvos. Nessa tela podemos exportar e importar desenvolvimentos que realizamos no Kibana, como dashboards, consultas e visualizações.

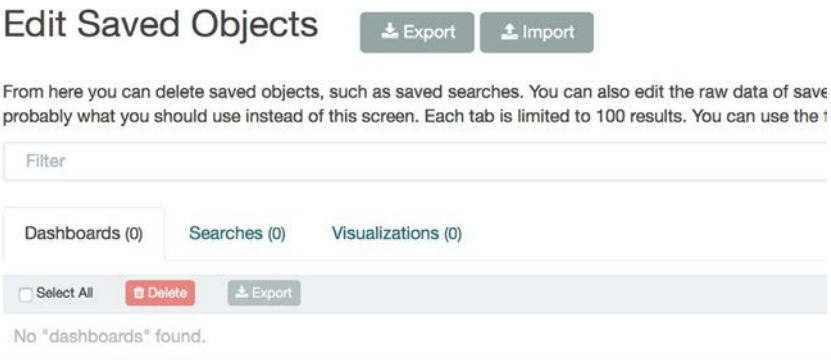



Figura 4.5: Tela de import/export de objetos

E, por fim, temos a tela *About*, que é somente uma tela onde podemos ver a versão do Kibana que temos instalada. Assim, concluímos a nossa configuração, então vamos começar a usar o nosso Kibana, começando pela aplicação *Discover*.

## 4.6 APLICAÇÕES DO KIBANA

### Discover

A aplicação 'Discover' permite que analisemos nossos dados em *real-time*, utilizando recursos como *drill-down* – análises que permitem *descer* a consulta até o último nível de informação, no nosso caso, os dados de um registro de evento de log – e filtragem dinâmica. Para entrarmos no Discover, vamos clicar na opção *Discover* no menu do topo do Kibana.

A próxima tela vai depender de quanto tempo faz que o seu pipeline não recebe dados. Por default, o Kibana filtra pelos dados dos últimos 15 minutos – repare no canto superior direito a mensagem *"Last 15 minutes"* –, então se fizer mais de 15 minutos desde que a última informação trafegou pelo nosso pipe, a tela será como a figura a seguir:

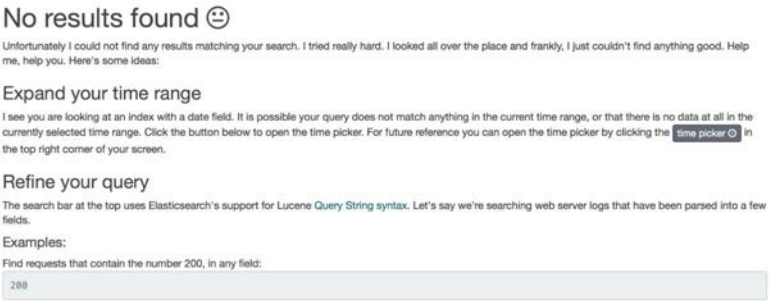



Figura 4.6: Tela do Kibana sem dados

Podemos resolver esse problema de duas formas: inserindo mais dados no pipeline, ou aumentando o range de pesquisa do Discover. Ao clicar no canto superior direito, uma aba se abre, onde vemos todas as opções de range disponíveis para a pesquisa. Vamos escolher 1 mês para o range, de modo a termos todos os nossos dados disponíveis.

Figura 4.7: Opções de range de datas do Kibana

Após selecionarmos o range, podemos ver a tela do Discover populada, onde temos diversas informações. Na parte superior da tela, temos um gráfico de barras que mostra a quantidade de eventos recebidos na linha do tempo, com o número total de eventos no canto superior direito do gráfico.

Na barra lateral a esquerda, podemos ver todos os campos selecionados e disponíveis para as nossas consultas – esses campos

são os que aparecem com opção para ordenação na tabela de eventos que vemos ao lado. Logo abaixo do gráfico, podemos ver os eventos de log, organizados por data. Por fim, podemos ver uma pequena barra editável logo acima do gráfico, com o caractere `*`. Esse campo é onde realizaremos nossas pesquisas, como veremos a seguir.

Figura 4.8: Tela do Discover

Vamos clicar na seta ao lado de um dos eventos. Ao fazer isso, veremos de forma expandida as informações do evento, seguindo o mapeamento que definimos nos capítulos anteriores:

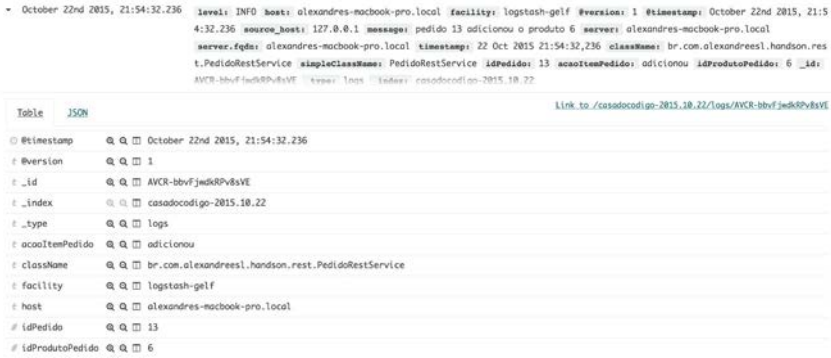



Figura 4.9: Drill-down de um evento de log

Agora, vamos experimentar adicionar alguns campos na tabela

de eventos. Vamos aproximar o mouse do campo `idPedido`. Podemos ver que um ícone 'add' (adicionar) irá aparecer, ao clicarmos nesse campo, é possível ver que adicionamos o campo a nossa tabela:

Figura 4.10: Adicionando campos a tabela de eventos

A título de curiosidade, vamos clicar no nome do campo em vez do ícone *Add*, por exemplo, do campo `StackTrace`. Nós teremos uma rápida visualização de contagem da quantidade de incidências do campo, junto com um *top* dos valores mais encontrados:

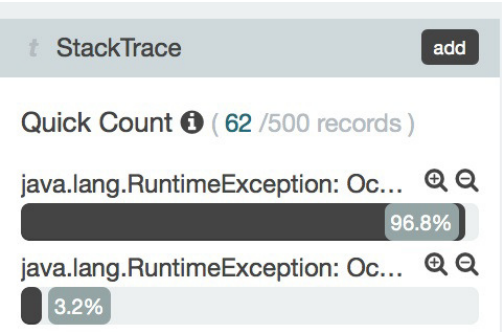



Figura 4.11: Contagem de incidências do campo

Vamos agora experimentar algumas consultas. Realizar consultas no Discover é muito fácil, basta seguir o mesmo padrão de sintaxe que vimos no capítulo *Dissecando a ELK - Elasticsearch*, quando vimos sobre query strings na seção *Consultas por múltiplos*

*campos (query_string).*

Por exemplo, se quisermos consultar todos as adições de itens nos carrinhos para os pedidos 11 e 13, basta fazermos uma consulta como a seguinte:

```
(idPedido:11 OR idPedido:13) AND acaoItemPedido:adicionou
```

A consulta produzirá um resultado como podemos ver adiante, onde já coloquei devidamente separadas as informações na visão tabular, para facilitar a leitura do leitor:

| Time | idPedido | acaoItemPedido |
|---|---|---|
| October 26th 2015, 22:56:28.841 | 13 | adicionou |
| October 26th 2015, 22:56:28.838 | 13 | adicionou |

Figura 4.12: Exemplo de consulta do Kibana

Ou se, por exemplo, quisermos realizar uma consulta de todas as remoções de itens que ocorreram nos carrinhos de compras, a consulta é mais simples ainda, sendo:

```
acaoItemPedido:removeu
```

Agora, vamos supor que essa é uma consulta muito comum, que vários usuários do nosso Kibana vão utilizar continuamente. Seria interessante termos essa consulta salva, para tirarmos a necessidade de digitá-la toda vez que formos usá-la, correto?

Para isso, basta salvarmos nossa consulta. Logo, vamos colocar a consulta no campo de texto e, em seguida, clicar no botão do diskete, como podemos ver:

Figura 4.13: Salvando uma consulta

Após clicarmos no ícone, o Kibana pedirá para nós darmos um nome para a consulta. Vamos chamá-la de `consultaremocaopedidos` e clicar no botão *Save*:

Figura 4.14: Dando um nome para a consulta

Depois, podemos ver que o nome da consulta aparece no canto direito, indicando que conseguimos salvá-la com sucesso:

Figura 4.15: Tela após salvar a consulta

E assim concluímos nosso estudo do Discover. Para obter mais informações sobre como montar consultas com o padrão da query string, você pode consultar o seguinte link: https://www.elastic.co/guide/en/elasticsearch/reference/current/query-dsl-query-string-query.html.

## Visualizações

Visualizações são gráficos, tabelas e outros tipos de visões que podemos montar em cima de nossos dados. Recapitulando o que vimos lá no capítulo *Introdução*, nosso chefe fez o seguinte pedido: *"Precisamos de uma dashboard para a diretoria que exiba, em tempo real, indicadores de consumidores com mais alterações de cadastro e quantidade de novos cadastrados por dia, além de um gráfico de crescimento de erros da interface, para que possamos colocar em um painel de monitoração no nosso service desk"*.

Naquela altura, ele estava pensando apenas na API de clientes, porém nós já montamos também a integração dos dados da nossa API de pedidos. Vamos surpreendê-lo e criar uma dashboard também de pedidos!

Vamos começar com o que ele pediu, começando com a construção de uma visão de top 5 dos consumidores que mais alteraram os seus cadastros. Para isso, vamos clicar no item *Visualize* no menu do topo do Kibana. Seremos recebidos pela tela inicial de criação das visualizações:

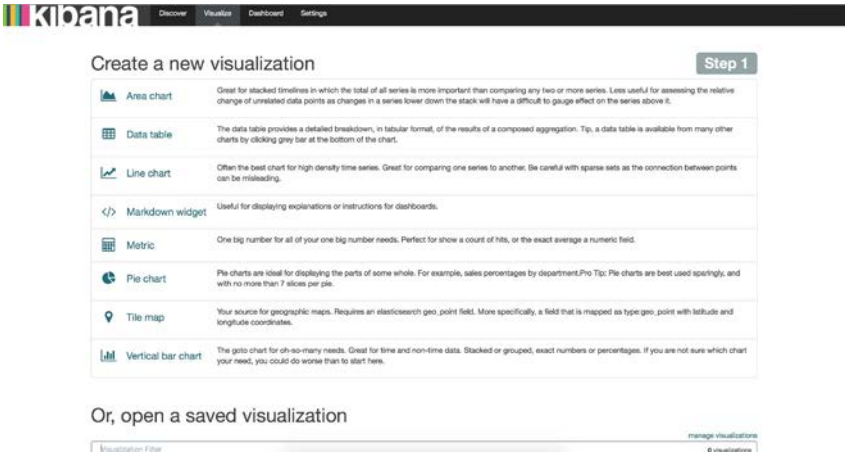



Figura 4.16: Tela de criação de visualizações

Vamos clicar no segundo item, *Data table* e, na próxima tela, clique em *From a new search* – se já tivéssemos uma consulta pronta do Discover, nesta tela poderíamos utilizá-la como base para a visualização. Seremos apresentados à tela de criação de visualizações, que podemos ver mais adiante.

Do lado direito da tela, podemos ver uma tabela onde serão plotadas as alterações que formos realizando na visualização, ao passo que do lado esquerdo podemos ver um painel onde realizamos as configurações. É possível ver no canto inferior

esquerdo, dentro da seção *Buckets*, 2 botões: um chamado `split rows` e outro chamado `split table`. Logo acima, temos uma seção *Metrics*, onde temos um valor de *count*. Basicamente, o que o painel quer dizer é que definimos a função a ser realizada para a montagem dos dados – a métrica –, e de que forma queremos que esses dados sejam separados, se em diferentes linhas de uma única tabela ou em várias tabelas.

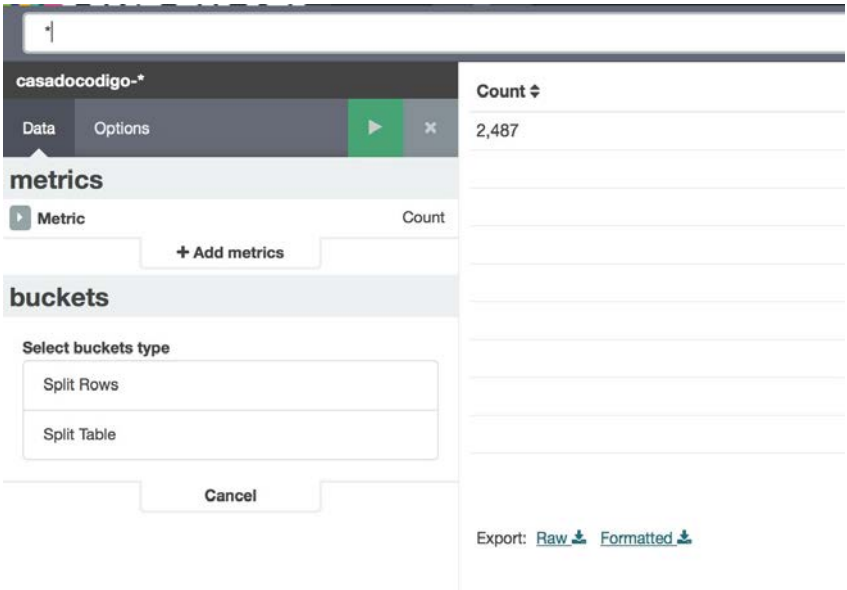



Figura 4.17: Tela inicial de construção da visualização

Vamos selecionar a opção `split rows`. Será aberto um novo formulário, onde podemos escolher a forma com que vamos agregar os dados das linhas e qual campo de nossos dados queremos usar. Vamos selecionar `term` como agregação e `idCliente` como o campo da agregação. A seguir, clicaremos no botão no final do painel, que adiciona uma subagregação.

Na subagregação, vamos selecionar `filters` como agregação e, no campo `filter 1`, vamos inserir o valor `acao:alterado`.

Vamos clicar na roda dentada para abrir as configurações avançadas e, na caixa de texto, vamos colocar um label para esse campo, chamado `alterações` . Na seção `metrics` , vamos deixar como está. A figura a seguir mostra os detalhes das configurações na seção `buckets` :

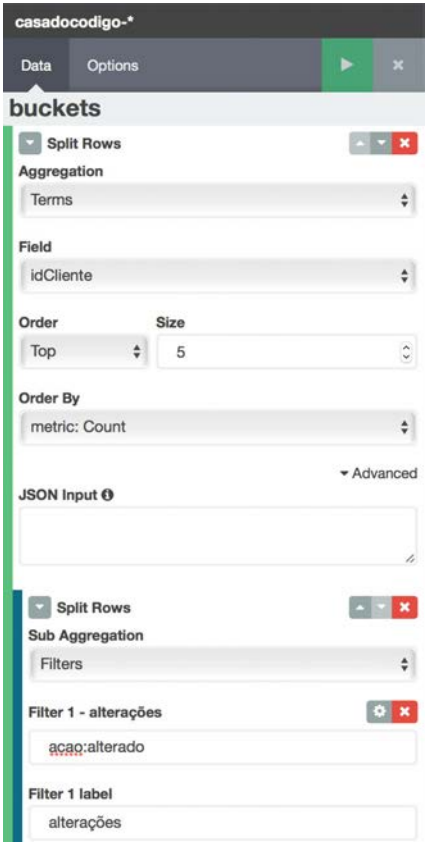



Figura 4.18: Configurações na seção 'buckets'

Após clicarmos no botão verde no canto superior esquerdo, podemos ver que a nossa tabela está pronta:

| Top 5 idCliente ⇕ Q | filters ⇕ | Count ⇕ |
|---|---|---|
| 1 | alterações | 100 |
| 4 | alterações | 100 |
| 2 | alterações | 50 |
| 0 | alterações | 50 |
| 6 | alterações | 100 |

Figura 4.19: Tela após salvar a consulta

Vamos agora salvar a nossa visualização. Para isso, basta fazer como fizemos para salvar a consulta, clicando no diskete no canto superior direito e fornecendo um nome para a visualização. Vamos chamá-la de *ranking de clientes alterados*.

O próximo item que o nosso chefe pediu é o gráfico de novos cadastros. Para isso, vamos clicar novamente em *Visualize* no menu do topo da tela para escolher uma nova visualização, e vamos desta vez escolher o gráfico de barras (*Vertical bar chart)*.

Seremos apresentados a uma tela análoga a de construção da *data table*, só que com uma visão de um gráfico de barras no lado direito. Para este gráfico, vamos selecionar `X-axis`, `Date Histogram` e o campo `@timestamp` na seção de `buckets`, e criar uma subagregaçao do tipo `filter`, com o valor `acao:inserido`. Na seção `metrics`, vamos trocar a métrica de `count` para `unique count`, e selecionar o field `idCliente`. A figura a seguir demonstra as configurações:

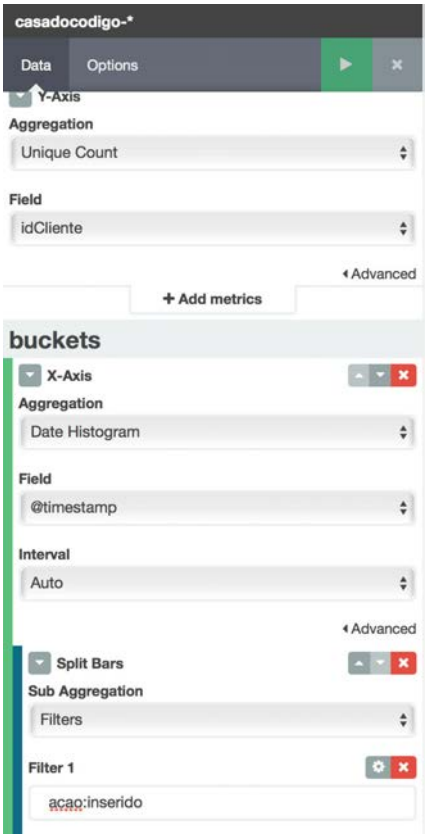



Figura 4.20: Configurações do relatório de novos cadastros

E após clicar no botão verde que efetiva as alterações, podemos ver nosso gráfico de barras. Vamos salvá-lo como *clientes novos por dia*.

Figura 4.21: Relatório de novos cadastros

Por fim, vamos terminar a demanda do chefe com o gráfico de erros da API de clientes. Para isso, vamos voltar para a tela de seleção de visualizações e selecionar a opção *line chart*.

Nessa visualização, vamos deixar `Y-axis` e `count` como métrica. Em `buckets`, vamos selecionar `X-axis`, `Date Histogram`, o campo `@timestamp` e o interval `Hourly`, e vamos selecionar uma subagregação de `filters` e inserir a query `level:ERROR and simpleClassName:ClienteRestService` no `field 1`. A figura adiante mostra o gráfico final, que salvaremos como *evolução de erros da API Clientes*.

Figura 4.22: Relatório de erros da API de Clientes

E assim concluímos os requisitos que o chefe nos pediu. A fim de não 'entediar' o leitor como repetidas seções que são praticamente idênticas umas às outras para construir o restante dos gráficos, deixo como desafio para o leitor criar os 3 últimos gráficos que vamos construir, que são:

- Um gráfico de linha de erros da API de pedidos, como o que fizemos agora;
- Uma tabela dos itens mais adicionados;
- Uma tabela dos itens mais removidos.

## Dashboards

Agora que já concluímos as visualizações, vamos criar a nossa dashboard para a entrega final. Vamos clicar no item *Dashboard* no menu do topo da interface. Seremos apresentados à tela de criação de dashboards, que podemos ver a seguir. Nela podemos ver os já

tradicionais botões de salvar, abrir e criar novas dashboards, uma caixa de busca onde podemos pesquisar as visualizações já criadas, e um botão de `+`, onde adicionamos as visualizações na nossa dashboard:

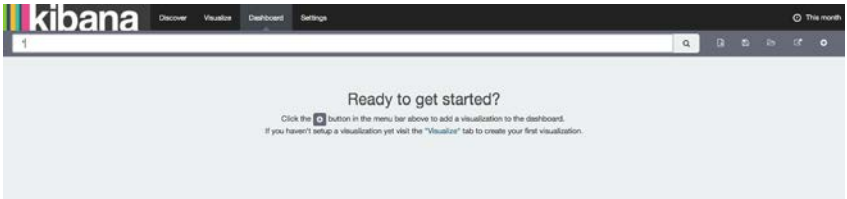



Figura 4.23: Tela inicial de dashboards

Vamos começar a sua construção. Para isso, vamos clicar no ícone `+`. A seguir, vamos selecionar o gráfico de novos cadastros, como na figura:

Figura 4.24: Selecionando gráfico de novos cadastros

Após selecionar a visualização, vamos perceber que temos criado o nosso gráfico em um `slot` de espaço na dashboard:

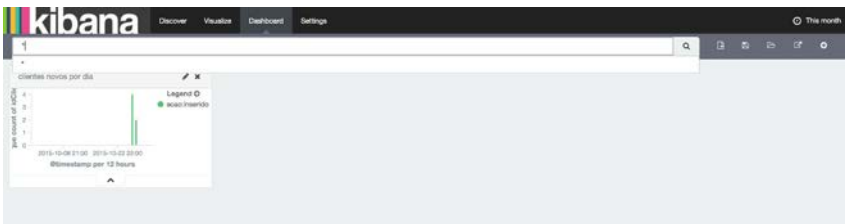



Figura 4.25: Tela com o gráfico incluído

Vamos aumentar um pouco o tamanho da área do gráfico. Para isso, basta aproximar com o mouse de uma das diagonais do slot da

dashboard, e arrastar com mouse para redimensionar a área. A seguir, vamos selecionar o gráfico de erros da API de Clientes. Isso criará outro slot na dashboard, com o outro gráfico:

Figura 4.26: Selecionando gráfico de erros da API de Clientes

Vamos redimensionar o gráfico de erros para ficar nas mesmas dimensões do outro gráfico, e vamos inverter a ordem dos gráficos. Para isso, basta selecionar o título do gráfico e arrastá-lo para a nova posição. Temos agora uma tela como:

Figura 4.27: Tela com os dois gráficos ajustados

E assim continuamos sucessivamente, até terminarmos de incluir todas as visualizações. Você pode montar a dashboard como desejar, mas se você quiser montar a dashboard da forma que eu fiz, segue a sua tela final:

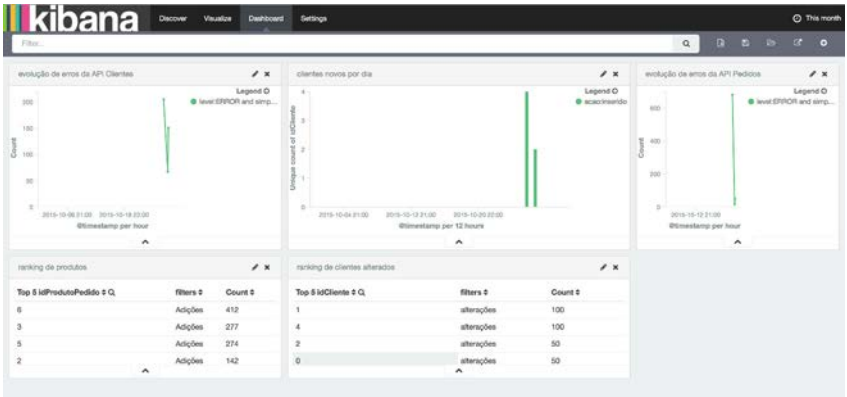



Figura 4.28: Dashboard final

Agora é só entregar para o chefe.

## 4.7 CONCLUSÃO

Concluímos o nosso tour pelo Kibana e pela famosa interface ELK. Como podemos ver, não foi difícil de criarmos dashboards que analisam dados em real-time, utilizando as informações de logs que temos disponíveis das nossas APIs. Experimente usar os scripts do JMeter e atualizar a dashboard enquanto eles executam, você vai ver que os dados são atualizados em segundos!

Espero que tenha conseguido transmitir para o leitor o poder dessas ferramentas, fomentando a vontade de experimentá-las em sua empresa. Agora, siga-me para o próximo capítulo, onde realizaremos mais práticas e veremos mais do poder de análise do Elasticsearch!

CAPÍTULO 5

# ELASTICSEARCH AVANÇADO

Agora que terminamos o nosso tour pela stack ELK, vamos começar a explorar recursos mais avançados do nosso Elasticsearch. Para este capítulo, vamos utilizar 2 práticas:

- Um pipeline de tweets oriundos do Twitter que vamos realizar diversas análises textuais;
- Por meio de uma massa de produtos e categorias, vamos realizar consultas pela hierarquia mercadológica dos produtos, em estilo *parent-child*.

Os códigos-fontes para as práticas deste capítulo podem ser encontrados em https://github.com/alexandreesl/livro-elasticsearch/tree/master/Capitulo%205.

Antes de começarmos a montagem das práticas, aprenderemos como funcionam as operações de CRUD (*Create-Remove-Update-Delete*) de índices e documentos no Elasticsearch, pois vamos utilizá-los em uma das práticas.

## 5.1 MANUTENÇÃO DE ÍNDICES

Na prática que desenvolvemos no capítulo anterior, nunca tivemos de manipular diretamente a criação e/ou alteração de

índices, pois o Logstash realizava esse trabalho para nós. Por baixo dos panos, porém, o Logstash estava trabalhando, realizando a criação dos índices e modificando os mapeamentos dos documentos contidos nele. Em cenários, porém, em que desejamos manipular de maneira mais direta os índices do indexador, é interessante para nós que saibamos como utilizar a API REST que o Elasticsearch nos provê.

## Criando um índice e um mapeamento

Vamos começar criando um índice. Este será usado na prática que vamos realizar neste capítulo, onde simularemos uma hierarquia mercadológica de produtos, que usaremos para realizar consultas posteriores.

Para criar o índice, executamos uma chamada `POST` para o endpoint do ELasticsearch, como vemos a seguir:

```
curl -XPOST 'localhost:9200/loja'
```

Por esse comando, criamos o nosso novo índice do Elasticsearch, chamado `loja` . Agora, para efeito de aprendizado, vamos criar um documento simples chamado de `documentoA` , que conterá 2 campos: o `campo1` do tipo texto, e o `campo2` do tipo inteiro. Este documento não será o utilizado na prática, que veremos na seção *Base de produtos no Elasticsearch* mais adiante neste capítulo.

```
curl -XPUT 'localhost:9200/loja/documentoA/_mapping' -d '{
  "documentoA" : {
    "properties" : {
      "campo1" : {"type" : "string"},
      "campo2" : {"type" : "integer"}
    }
} }'
```

Vamos agora checar os mapeamentos do nosso índice para

vermos se o comando criou o nosso mapeamento adequadamente. Para isso, usamos o mesmo comando que aprendemos no capítulo *Dissecando a ELK – Elasticsearch*:

```
curl -XGET 'localhost:9200/loja/_mapping?pretty'
```

O comando deve produzir um JSON como o seguinte, onde podemos ver o mapeamento do nosso documento chamado `documentoA`:

```
{
  "loja" : {
    "mappings" : {
      "documentoA" : {
        "properties" : {
          "campo1" : {
            "type" : "string"
          },
          "campo2" : {
            "type" : "integer"
          }
        }
      }
    }
  }
}
```

## Modificando um mapeamento

Vamos supor agora que detectemos que, na verdade, faltou um campo no nosso mapeamento, chamado `campo3`, também do tipo texto. Para adicionarmos o campo, podemos executar o seguinte comando:

```
curl -XPUT 'localhost:9200/loja/documentoA/_mapping' -d '{
  "documentoA" : {
    "properties" : {
      "campo3" : { "type" : "string" }
    }
} }'
```

Outra alteração que realizaremos é incluir um campo chamado `campo4`. Você pode estar pensando: *"Nossa, mas que trabalho criar os campos um por um!"*

Realmente, essa é uma forma muito complicada de se criar o *mapping*. Vamos ver uma outra forma de realizar a criação dos mapeamentos, onde já criamos em uma única chamada tanto o índice quanto o documento. Para isso, primeiro, vamos remover o documento que criamos, com o comando:

```
curl -XDELETE 'http://localhost:9200/loja/documentoA'
```

> **Importante**
>
> Infelizmente, a partir do Elasticsearch 2.x, **NÃO** é mais possível remover `document types`! Nesse cenário, a recomendação da Elastic é criar um novo índice com os mapeamentos corretos. Sendo assim, o comando anterior só funcionará no Elasticsearch 1.x.

Vamos agora remover o índice. Lembre-se de que no capítulo *Dissecando a ELK – Elasticsearch*, antes de criarmos o nosso padrão de índice com o prefixo `casadocodigo`, criamos um índice com o prefixo `testes`? Pois bem, vamos agora apagar aquele índice de teste que criamos, visto que não vamos mais usá-lo.

Para começar, vamos executar o seguinte comando, que listará todos os índices dentro do indexador:

```
curl 'localhost:9200/_cat/indices?v'
```

Nós receberemos da console um output parecido com o da figura:

```
bin — alexandrelourenco@Alexandres-MacBook-Pro — ..ash-1.5.4/bin — -zsh...
→  bin  curl 'localhost:9200/_cat/indices?v'
health status index                    pri rep docs.count docs.deleted store.size
 pri.store.size
yellow open   casadocodigo-2015.10.28   5   1          1            0      7.1kb
          7.1kb
yellow open   casadocodigo-2015.10.27   5   1       3141            0        2mb
            2mb
yellow open   .kibana                   1   1          9            1       14kb
           14kb
yellow open   twitter-2015.11.02        5   1      20664            0     12.9mb
         12.9mb
yellow open   twitter-2015.11.01        5   1      17452            0     11.3mb
         11.3mb
yellow open   loja                      5   1          0            0       720b
           720b
yellow open   testes-2015.10.27         5   1          1            0         7kb
            7kb
→  bin
```

Figura 5.1: Listagem dos índices criados no Elasticsearch

Agora que já temos listados os nossos índices, podemos ver o nome do índice que desejamos excluir – no meu caso, o nome do índice é `testes-2015.10.27`. Para removermos esse índice, basta executarmos o comando:

```
curl -XDELETE 'http://localhost:9200/testes-2015.10.27/'
```

Após a execução do comando, poderemos ver pelo console do Elasticsearch que o índice foi removido, pois aparecerá uma mensagem como a seguinte:

```
[2015-11-02 22:41:33,545][INFO ][cluster.metadata         ] [Dr. M
arla Jameson] [testes-2015.10.27] deleting index
```

E é claro, se executarmos novamente a listagem de índices, poderemos ver que ele foi removido com sucesso. Vamos agora remover o índice `loja`. Para isso, basta executar outro comando, análogo como o anterior:

```
curl -XDELETE 'http://localhost:9200/loja/'
```

A seguir, vamos executar o próximo comando, em que criamos o índice, documento e seus 4 campos em uma única chamada:

```
curl -XPUT 'localhost:9200/loja' -d '{
  "mappings": {
    "documentoA": {
      "properties": {
        "campo1": {
          "type": "string"
        },
        "campo2": {
          "type": "integer"
        },
        "campo3": {
          "type": "string"
        },
        "campo4": {
          "type": "string"
        }
      }
    }
  }
}'
```

Após executarmos novamente o comando que exibe os mapeamentos do índice – `curl -XGET 'localhost:9200/loja/_mapping?pretty'` –, poderemos ver que toda a estrutura foi criada com sucesso. Porém, detectamos que um de nossos campos foi criado com o tipo errado: o `campo2` também era para ter sido criado como texto! Vamos tentar modificar o tipo do campo com o comando:

```
curl -XPUT 'localhost:9200/loja/documentoA/_mapping' -d '{
  "documentoA" : {
    "properties" : {
      "campo2" : { "type" : "string" }
    }
} }'
```

Ao tentarmos executar a alteração, porém, nos deparamos com a seguinte mensagem de erro:

```
{"error":"MergeMappingException[Merge failed with failures {[mappe
r [campo2] of different type, current_type [integer], merged_type
[string]]}]","status":400}
```

A razão para isso é que certos tipos de alterações não são permitidas pela API do Elasticsearch a fim de garantir a consistência dos dados, visto que o Elasticsearch não fará a alteração de tipo para os documentos que foram indexados anteriormente. Atente-se que esta "barreira" não impede que o mesmo campo seja criado com tipos diferentes em diferentes mapeamentos de documentos, o que significa que a nossa recomendação de atenção quanto à tipagem de campos continua valendo.

Sumarizando, temos a seguinte lista de ações que podem ou não ser realizadas:

- **Pode ser realizado:**
  - Adição de campos;
  - Adição de analisadores (veremos isso em detalhes na seção *Aprofundando em analisadores textuais*).
- **Não pode ser realizado:**
  - Mudança de tipos de campos;
  - Mudança da propriedade `store` de `true` para `false`, e vice-versa. O campo `store` pode ser usado para indicar se o valor do campo deve ou não ser armazenado no Elasticsearch para retorno nas consultas – embora ele ainda possa ser obtido dentro do atributo `_source` do retorno;
  - Troca de um analisador depois que documentos já foram indexados no índice.

Agora que concluímos a nossa passagem geral pelas ações que podemos realizar nos índices, vamos começar a ver o que podemos

realizar nos documentos.

## 5.2 MANUTENÇÃO DE DOCUMENTOS

Agora que temos criado o nosso tipo de documento `documentoA`, começaremos inserindo um documento:

```
curl -XPOST 'localhost:9200/loja/documentoA' -d '{ "campo1" : "est
e e o valor do campo 1" , "campo2" : 123 , "campo3" : "este e o va
lor do campo 3" , "campo4" : "este e o valor do campo 4" }'
```

Após a execução do comando, vamos receber uma mensagem no console como a seguinte, onde podemos observar informações como o índice e o document type utilizados na inserção, um id gerado automaticamente pela ferramenta e uma flag indicativa de sucesso da operação, indicando que a inserção do documento foi feita com sucesso:

```
{"_index":"loja","_type":"documentoA","_id":"AVDVkBQ8pX7KXgHP6zNq",
"_version":1,"created":true}
```

Em um sistema que manipula diretamente os documentos do Elasticsearch, é muito importante que guardemos esses ids, pois os usaremos para as operações de alteração e exclusão, que veremos a seguir.

Vamos agora alterar o documento, mudando o valor do `campo1`. Para realizarmos essa operação, realizamos um comando como:

```
curl -XPUT 'localhost:9200/loja/documentoA/AVDVkBQ8pX7KXgHP6zNq' -d
 '{ "campo1" : "novo valor do campo 1" , "campo2" : 123 , "campo3"
 : "este e o valor do campo 3" , "campo4" : "este e o valor do cam
po 4" }'
```

**Importante**

Nos comandos de atualização, é preciso repassar os valores de todos os campos para o Elasticsearch, do contrário, ele interpretará que queremos remover os valores dos campos não informados!

Após a execução, teremos uma resposta análoga à seguinte. Atente-se para o campo `version`, que subiu – falaremos mais sobre isso na próxima seção.

```
{"_index":"loja","_type":"documentoA","_id":"AVDVkBQ8pX7KXgHP6zNq",
"_version":2,"created":false}
```

Antes de partirmos para a exclusão, vamos somente testar uma simples consulta de todos os documentos contidos no índice, como objetivo de constatarmos se o nosso documento se encontra corretamente no índice:

```
curl -XGET 'http://localhost:9200/loja/_search'
```

O retorno confirma que o nosso documento foi armazenado com sucesso:

```
{"took":2,"timed_out":false,"_shards":{"total":5,"successful":5,"f
ailed":0},"hits":{"total":1,"max_score":1.0,"hits":[{"_index":"loj
a","_type":"documentoA","_id":"AVDVkBQ8pX7KXgHP6zNq","_score":1.0,
"_source":{ "campo1" : "novo valor do campo 1" , "campo2" : 123 ,
"campo3" : "este e o valor do campo 3" , "campo4" : "este e o valo
r do campo 4" }}]}}
```

Finalizando, vamos remover o documento pelo comando a seguir:

```
curl -XDELETE http://localhost:9200/loja/documentoA/AVDVkBQ8pX7KXg
HP6zNq
```

Isso vai produzir um retorno como o seguinte:

```
{"found":true,"_index":"loja","_type":"documentoA","_id":"AVDVkBQ8
pX7KXgHP6zNq","_version":3}
```

Se executarmos novamente a consulta, vamos constatar que o índice não retorna mais nenhum documento, ou seja, a exclusão foi feita com sucesso. Reparou, entretanto, na estranha 'versão 3' que foi gerada do documento? Isso se deve às características com que o Elasticsearch trabalha as remoções e atualizações de documentos, como falaremos a seguir.

## Alteração e remoção de documentos no Elasticsearch

Conforme já falamos anteriormente, por baixo dos panos do Elasticsearch, temos o Apache Lucene. Isso significa que, quando estamos criando documentos, na verdade estamos gerando dados dentro de índices do Lucene. Dentro dessa estrutura, NÃO é possível atualizar e/ou remover dados, uma vez que eles já tenham sido gravados no Lucene. É isso mesmo: quando estamos atualizando ou removendo um documento, na verdade não estamos fazendo isso, mas sim gerando uma nova versão dele.

No caso da exclusão e da versão antiga no caso da atualização de documentos, esses documentos são marcados como desabilitados, ou seja, é feita uma exclusão lógica. Um ponto importante é que, embora não vejamos o documento excluído nos resultados, ele ainda está lá, e o Elasticsearch é obrigado a passar por ele durante a sua consulta, apenas para descartá-lo devido a ele estar marcado para deleção.

Mas como esse "lixo" é removido? Para isso, o Elasticsearch conta com uma operação chamada *segment merge* (mescla de

segmentos). Nela é criado um novo fragmento (*shard*) onde todos os documentos ativos de um ou mais fragmentos antigos são copiados para dentro de si. Após a cópia ter sido completada, todos os fragmentos antigos são fisicamente deletados do Elasticsearch.

O que acontece se temos um índice com muitas deleções de documentos? Além da grande quantidade de disco gasta com armazenamento de documentos que não são mais desejáveis, temos um maior consumo de CPU e memória no processamento desse volume "morto" de dados, além de maior lentidão nas consultas, devido a grande quantidade de documentos desabilitados que o Elasticsearch deve percorrer em cada consulta! Além disso, análogo ao famoso Garbage Collector do Java, essa operação é bastante dispendiosa, sendo portanto uma operação que não desejamos que ocorra constantemente nas nossas soluções.

Assim, temos de ter muito cuidado ao usar soluções que englobam o uso de muitas exclusões de documentos, pois possivelmente teremos problemas de performance. Uma melhor solução é tentarmos soluções que isolem os documentos que não são mais necessários em índices apartados, deixando que o processo de expurga seja responsável pela exclusão dos dados.

Agora que já aprendemos como realizar operações básicas sobre os índices e documentos, vamos começar a montar os exercícios práticos!

## 5.3 MONTANDO OS EXERCÍCIOS PRÁTICOS

### Pipeline do Twitter

Vamos começar montando o nosso pipeline do Twitter. Para isso, vamos reaproveitar o nosso arquivo de configuração do Logstash do capítulo *Dissecando a ELK – Elasticsearch*, reutilizando

a configuração de conexão com o Elasticsearch, porém mudando o seu índice para o prefixo `twitter-`, a fim de não misturarmos os dados com a nossa prática anterior. Vamos também configurar o plugin do Twitter.

Antes de instalar o plugin, é preciso ter uma conta no Twitter e, em seguida, criar uma aplicação do Twitter, que permitirá ao nosso plugin fazer a conexão com a stream pública do Twitter. A aplicação pode ser criada em https://apps.twitter.com/app/new.

Após criar o aplicativo, entre na tela de propriedades do aplicativo, na aba *Keys and access tokens*, e obtenha as chaves:

- Consumer Key (API Key);
- Consumer Secret (API Secret);
- Access Token;
- Access Token Secret.

Após a criação da aplicação, vamos criar o arquivo de configuração `twitterpipeline.conf`, onde inseriremos as chaves de configuração do aplicativo – substitua as suas chaves nos campos em `...`, as chaves de access token vão nos campos com o prefixo `oauth` – e configuraremos o novo índice com o prefixo `twitter-`:

```
input {
      twitter {
        consumer_key => ...
        consumer_secret => ...
        keywords => ["coca cola","java","elasticsearch","amazon"]
        oauth_token => ...
        oauth_token_secret => ...
    }
}
```

```
output {
      stdout { codec => rubydebug }
```

```
      elasticsearch {
            hosts => [ "localhost:9200" ]
            index => "twitter-%{+YYYY.MM.dd}"
        }
}
```

Após inicializar o pipeline, poderemos ver a nossa console continuamente inputando mensagens de tweets da stream do Twitter, como podemos ver na figura a seguir. Na nossa configuração, configuramos que queremos todos os tweets com as palavras `coca cola` , `java` , `elasticsearch` e `amazon` . Vamos deixar o pipeline rodando por cerca de meia hora para termos uma boa base textual.

Figura 5.2: Console do pipeline do Twitter

O arquivo `twitterpipeline.conf` também pode ser encontrado no GitHub, no endereço no ínicio deste capítulo.

Agora, vamos seguir para a montagem da outra prática do nosso capítulo.

## Base de produtos no Elasticsearch

Para esta prática, vamos construir uma hierarquia mercadológica, na qual teremos produtos e categorias. Começaremos criando o mapeamento do documento onde cadastraremos as categorias pelo comando:

```
curl -XPUT 'localhost:9200/loja/categoria/_mapping' -d '{
  "categoria" : {
    "properties" : {
      "nome" : {"type" : "string"}
    }
} }'
```

A seguir, criamos o mapeamento do documento de produtos. Repare na propriedade `_parent`, que utilizamos para fazer a referência pai para os nossos documentos de categorias:

```
curl -XPUT 'localhost:9200/loja/produto/_mapping' -d '{
  "produto" : {
  "_parent" : { "type" : "categoria" },
    "properties" : {
      "sku" : {"type" : "integer"},
      "nome" : {"type" : "string"},
      "descricao" : {"type" : "string"},
      "marca" : {"type" : "string"},
      "estoque" : {"type" : "integer"}
    }
} }'
```

Na versão 2.1.0 do Elasticsearch, há um bug que não nos permite a criação de mapeamentos de documentos filhos após o documento pai ter sido criado. Se o leitor estiver recebendo um erro como `Can't add a parent field that points to an already existing type`, basta inverter a ordem dos comandos, criando o mapeamento de produto antes do de categoria.

Agora que temos os mapeamentos criados, vamos começar o cadastramento das categorias e produtos. Todos os comandos – inclusive os de mapeamentos anteriores – podem ser encontrados em `comandoselasticsearch.txt` , dentro do repositório citado no começo deste capítulo.

Se o leitor estiver usando um sistema Unix/Linux, basta renomear o arquivo como um shell script, e executá-lo para efetuar a carga com uma única execução. Para as categorias, vamos cadastrar uma série de categorias, conforme a série de comandos a seguir. Repare que, nesse caso, declaramos explicitamente o id, dentro da URL, em vez de deixar o Elasticsearch gerar os IDs para nós. A razão para isto é que precisamos desses IDs para fazer os vínculos com os documentos de produtos.

```
curl -XPOST 'localhost:9200/loja/categoria/Geladeiras' -d '{ "nome
" : "Geladeiras" }'
curl -XPOST 'localhost:9200/loja/categoria/Fogões' -d '{ "nome" :
"Fogões" }'
curl -XPOST 'localhost:9200/loja/categoria/Ar-condicionados' -d '{
 "nome" : "Ar-condicionados" }'
curl -XPOST 'localhost:9200/loja/categoria/Aspiradores' -d '{ "nom
e" : "Aspiradores" }'
curl -XPOST 'localhost:9200/loja/categoria/Masculino' -d '{ "nome"
 : "Masculino" }'
curl -XPOST 'localhost:9200/loja/categoria/Feminino' -d '{ "nome"
: "Feminino" }'
curl -XPOST 'localhost:9200/loja/categoria/Lar' -d '{ "nome" : "La
r"  }'
```

**Importante**

Muito cuidado ao trabalhar com relacionamentos de *child-parent* na hora do cadastramento, pois se cadastrarmos um valor incorreto como *parent*, o Elasticsearch não acusará o problema, ocasionando o problema de termos documentos órfãos!

E finalmente, vamos cadastrar os produtos, cadastrando-os sob diversas categorias:

```
curl -XPOST 'localhost:9200/loja/produto?parent=Geladeiras' -d '{
"sku" : 234565678,"nome":"produto 1", "marca": "marcaA" ,"descrica
o":"esta é a descrição do produto 1","estoque":23 }'
curl -XPOST 'localhost:9200/loja/produto?parent=Fogões' -d '{ "sku
" : 234567288,"nome":"produto 12", "marca": "marcaA" ,"descricao":
"esta é a descrição do produto 12","estoque":3 }'
curl -XPOST 'localhost:9200/loja/produto?parent=Fogões' -d '{ "sku
" : 234567289,"nome":"produto 13", "marca": "marcaB","descricao":"
esta é a descrição do produto 13","estoque":7 }'
curl -XPOST 'localhost:9200/loja/produto?parent=Fogões' -d '{ "sku
" : 234567290,"nome":"produto 14", "marca": "marcaC","descricao":"
esta é a descrição do produto 14","estoque":11 }'
curl -XPOST 'localhost:9200/loja/produto?parent=Fogões' -d '{ "sku
" : 234567291,"nome":"produto 15", "marca": "marcaB","descricao":"
esta é a descrição do produto 15","estoque":16 }'
curl -XPOST 'localhost:9200/loja/produto?parent=Fogões' -d '{ "sku
" : 234567292,"nome":"produto 16", "marca": "marcaA","descricao":"
esta é a descrição do produto 16","estoque":20 }'
curl -XPOST 'localhost:9200/loja/produto?parent=Fogões' -d '{ "sku
" : 234567293,"nome":"produto 17", "marca": "marcaA","descricao":"
esta é a descrição do produto 17","estoque":33 }'
curl -XPOST 'localhost:9200/loja/produto?parent=Aspiradores' -d '{
 "sku" : 147567288,"nome":"produto 33", "marca": "marcaA","descric
ao":"esta é a descrição do produto 33","estoque":67 }'
curl -XPOST 'localhost:9200/loja/produto?parent=Aspiradores' -d '{
 "sku" : 147567289,"nome":"produto 34", "marca": "marcaA","descric
ao":"esta é a descrição do produto 34","estoque":23 }'
curl -XPOST 'localhost:9200/loja/produto?parent=Aspiradores' -d '{
 "sku" : 147567290,"nome":"produto 35", "marca": "marcaA","descric
```

```
ao":"esta é a descrição do produto 35","estoque":53 }'
curl -XPOST 'localhost:9200/loja/produto?parent=Ar-condicionados'
-d '{ "sku" : 234561828,"nome":"produto 4", "marcaA": "marcaB","de
scricao":"esta é a descrição do produto 4","estoque":3 }'
curl -XPOST 'localhost:9200/loja/produto?parent=Masculino' -d '{ "
sku" : 231007288,"nome":"produto 5", "marca": "marcaF","descricao"
:"esta é a descrição do produto 5","estoque":35 }'
curl -XPOST 'localhost:9200/loja/produto?parent=Lar' -d '{ "sku" :
 234567900,"nome":"produto 6", "marca": "marcaH","descricao":"esta
 é a descrição do produto 6","estoque":89 }'
```

Repare que, nos scripts anteriores, usamos o query parameter `parent`, onde indicamos o documento pai dos documentos que estamos cadastrando, assim concluindo o nosso cadastro. Mas será que cadastramos nossos produtos e categorias da melhor forma possível? É o que veremos a seguir.

## Utilizando routing customizado no Elasticsearch

Em nossas consultas, o mais comum é que pesquisemos apenas dentro de uma categoria. Por exemplo, se estivermos pesquisando produtos de eletro, não faz muito sentido que precisemos dos produtos de vestuário.

Isso pode não fazer muita diferença na nossa pequena massa, mas faz uma grande diferença em grandes massas a serem analisadas, em que, da forma que desenvolvemos, o indexador deverá pesquisar por todos os documentos do índice. Podemos resolver isso utilizando o recurso `_routing`, que veremos a seguir!

Por default, o Elasticsearch gera uma hash a partir do ID do documento, que no nosso caso é gerado automaticamente, e depois o documento é alocado em um shard de acordo com a hash gerada. Quando utilizamos o `_routing`, essa regra é modificada para a seguinte fórmula:

```
_shard_num = hash(_routing) % num_primary_shards_
```

Assim, temos uma indexação e busca mais adequada, de acordo com os valores que passarmos. Para isso, basta acrescentar o campo `routing` dentro das query parameters dos comandos de inclusão das categorias e produtos, onde modelamos os cadastros para que os shards reflitam as categorias-chave dentro da nossa hierarquia mercadológica, como podemos ver:

```
curl -XPOST 'localhost:9200/loja/categoria/Geladeiras?routing=Elet
ro' -d '{ "nome" : "Geladeiras" }'
curl -XPOST 'localhost:9200/loja/categoria/Fogões?routing=Eletro'
-d '{ "nome" : "Fogões" }'
curl -XPOST 'localhost:9200/loja/categoria/Ar-condicionados?routin
g=Eletro' -d '{ "nome" : "Ar-condicionados" }'
curl -XPOST 'localhost:9200/loja/categoria/Aspiradores?routing=Ele
tro' -d '{ "nome" : "Aspiradores" }'
curl -XPOST 'localhost:9200/loja/categoria/Masculino?routing=Vestu
ário' -d '{ "nome" : "Masculino" }'
curl -XPOST 'localhost:9200/loja/categoria/Feminino?routing=Vestuá
rio' -d '{ "nome" : "Feminino" }'
curl -XPOST 'localhost:9200/loja/categoria/Lar?routing=Lar' -d '{
"nome" : "Lar"  }'
curl -XPOST 'localhost:9200/loja/produto?parent=Geladeiras&routing
=Eletro' -d '{ "sku" : 234565678,"nome":"produto 1", "marca": "mar
caA","descricao":"esta é a descrição do produto 1","estoque":23 }'
curl -XPOST 'localhost:9200/loja/produto?parent=Fogões&routing=Ele
tro' -d '{ "sku" : 234567288,"nome":"produto 12", "marca": "marcaA
","descricao":"esta é a descrição do produto 12","estoque":3 }'
curl -XPOST 'localhost:9200/loja/produto?parent=Fogões&routing=Ele
tro' -d '{ "sku" : 234567289,"nome":"produto 13", "marca": "marcaB
","descricao":"esta é a descrição do produto 13","estoque":7 }'
curl -XPOST 'localhost:9200/loja/produto?parent=Fogões&routing=Ele
tro' -d '{ "sku" : 234567290,"nome":"produto 14", "marca": "marcaC
","descricao":"esta é a descrição do produto 14","estoque":11 }'
curl -XPOST 'localhost:9200/loja/produto?parent=Fogões&routing=Ele
tro' -d '{ "sku" : 234567291,"nome":"produto 15", "marca": "marcaB
","descricao":"esta é a descrição do produto 15","estoque":16 }'
curl -XPOST 'localhost:9200/loja/produto?parent=Fogões&routing=Ele
tro' -d '{ "sku" : 234567292,"nome":"produto 16", "marca": "marcaA
","descricao":"esta é a descrição do produto 16","estoque":20 }'
curl -XPOST 'localhost:9200/loja/produto?parent=Fogões&routing=Ele
tro' -d '{ "sku" : 234567293,"nome":"produto 17", "marca": "marcaA
","descricao":"esta é a descrição do produto 17","estoque":33 }'
curl -XPOST 'localhost:9200/loja/produto?parent=Aspiradores&routin
g=Eletro' -d '{ "sku" : 147567288,"nome":"produto 33", "marca": "m
arcaA","descricao":"esta é a descrição do produto 33","estoque":67
```

```
 }'
curl -XPOST 'localhost:9200/loja/produto?parent=Aspiradores&routin
g=Eletro' -d '{ "sku" : 147567289,"nome":"produto 34", "marca": "m
arcaA","descricao":"esta é a descrição do produto 34","estoque":23
 }'
curl -XPOST 'localhost:9200/loja/produto?parent=Aspiradores&routin
g=Eletro' -d '{ "sku" : 147567290,"nome":"produto 35", "marca": "m
arcaA","descricao":"esta é a descrição do produto 35","estoque":53
 }'
curl -XPOST 'localhost:9200/loja/produto?parent=Ar-condicionados&r
outing=Eletro' -d '{ "sku" : 234561828,"nome":"produto 4", "marca"
: "marcaB","descricao":"esta é a descrição do produto 4","estoque"
:3 }'
curl -XPOST 'localhost:9200/loja/produto?parent=Masculino&routing=
Vestuário' -d '{ "sku" : 231007288,"nome":"produto 5", "marca": "m
arcaF","descricao":"esta é a descrição do produto 5","estoque":35
}'
curl -XPOST 'localhost:9200/loja/produto?parent=Lar&routing=Lar' -d
 '{ "sku" : 234567900,"nome":"produto 6", "marca": "marcaH","descr
icao":"esta é a descrição do produto 6","estoque":89 }'
```

Pronto! Temos agora a nossa massa devidamente carregada. Agora que concluímos os preparativos para o nosso estudo deste capítulo, vamos começar pelas consultas na massa de produtos que acabamos de montar, usando o esquema de roteamento que definimos.

## 5.4 REALIZANDO CONSULTAS PARENT-CHILD

### Query do tipo 'has_child'

Vamos agora testar queries dentro da nossa árvore de categorias e produtos. Vamos supor que gostaríamos de saber todas as categorias cujos produtos sejam da marca `marcaA`. Para isso, basta usarmos a query do tipo `has_child`, como podemos ver adiante. Nessa query, informamos no atributo `type` o tipo do documento filho que queremos filtrar, e no atributo `query` informamos a query cujos filhos queremos filtrar, obtendo em seguida os seus

respectivos pais:

```
curl -XGET 'localhost:9200/loja/_search?pretty=true&routing=Eletro'
 -d '{
  "query" : {
    "has_child" : {
        "type" : "produto",
        "query": {
          "query_string" : {
              "query": "marca:marcaA"
          }
        }
     }
  }
}'
```

Essa query produzirá um resultado como o seguinte:

```
{
  "took" : 1,
  "timed_out" : false,
  "_shards" : {
    "total" : 1,
    "successful" : 1,
    "failed" : 0
  },
  "hits" : {
    "total" : 3,
    "max_score" : 1.0,
    "hits" : [ {
      "_index" : "loja",
      "_type" : "categoria",
      "_id" : "Geladeiras",
      "_score" : 1.0,
      "_source":{ "nome" : "Geladeiras" }
    }, {
      "_index" : "loja",
      "_type" : "categoria",
      "_id" : "Foges"  ,
      "_score" : 1.0,
      "_source":{ "nome" : "Fogões" }
    }, {
      "_index" : "loja",
      "_type" : "categoria",
      "_id" : "Aspiradores",
      "_score" : 1.0,
```

```
      "_source":{ "nome" : "Aspiradores" }
    } ]
  }
}
```

Reparou que nos dados relacionados aos shards da consulta, o Elasticsearch informou que percorreu apenas 1 shard? Isso é devido ao atributo `routing` que informamos nos parâmetros da nossa query. Isso faz com que tenhamos a já citada agilidade nas nossas consultas, já que o indexador pesquisará apenas pelos shards que nos interessam.

## Query do tipo 'has_parent'

Na query `has_parent`, temos exatamente o inverso da query `has_child`: nessa query, obtemos documentos filhos a partir de pesquisas que fazemos nos documentos pais. Por exemplo, vamos supor que queiramos buscar por todos os produtos cadastrados dentro da categoria `Aspiradores`. Podemos realizar essa consulta da seguinte forma:

```
curl -XGET 'localhost:9200/loja/_search?pretty=true&routing=Eletro'
 -d '{
  "query" : {
    "has_parent" : {
        "parent_type" : "categoria",
        "query": {
          "query_string" : {
              "query": "nome:Aspiradores"
          }
        }
     }
  }
}'
```

Essa consulta gerará os seguintes resultados, onde vemos que os resultados seguem o nosso esperado, retornando os 3 produtos cadastrados dentro da categoria `Aspiradores`:

```
{
  "took" : 13,
  "timed_out" : false,
  "_shards" : {
    "total" : 1,
    "successful" : 1,
    "failed" : 0
  },
  "hits" : {
    "total" : 3,
    "max_score" : 1.0,
    "hits" : [ {
      "_index" : "loja",
      "_type" : "produto",
      "_id" : "AVD0W8tgGEZKX2UpmuwA",
      "_score" : 1.0,
      "_source":{ "sku" : 147567288,"nome":"produto 33", "marca":
"marcaA","descricao":"esta é a descrição do produto 33","estoque":
67 }
    }, {
      "_index" : "loja",
      "_type" : "produto",
      "_id" : "AVD0W8trGEZKX2UpmuwB",
      "_score" : 1.0,
      "_source":{ "sku" : 147567289,"nome":"produto 34", "marca":
"marcaA","descricao":"esta é a descrição do produto 34","estoque":
23 }
    }, {
      "_index" : "loja",
      "_type" : "produto",
      "_id" : "AVD0W8t1GEZKX2UpmuwC",
      "_score" : 1.0,
      "_source":{ "sku" : 147567290,"nome":"produto 35", "marca":
"marcaA","descricao":"esta é a descrição do produto 35","estoque":
53 }
    } ]
  }
}
```


E assim concluímos nosso estudo sobre consultas em estruturas pai-filho. Como podemos ver com estes simples exemplos, se trata de uma estrutura de consultas poderosa, que vale a pena ser explorada em soluções que envolvam massas de dados associadas a alguma estrutura taxonômica.

Agora, vamos prosseguir nossos estudos, usando a massa de

dados oriundos do Twitter que obtemos no pipeline que criamos a pouco.

## 5.5 APROFUNDANDO EM ANALISADORES TEXTUAIS

No capítulo *Dissecando a ELK – Elasticsearch*, aprendemos um pouco sobre os analisadores, estes que são compostos dos seguintes componentes, que efetuam a análise textual dos dados: *Character filters*, *Tokenizer* e *TokenFilters*.

Por default, sem que façamos nenhuma configuração adicional, os dados textuais já passam por um analisador textual default. Além dele, podemos encontrar outros tipos, como os seguintes:

- **Whitespace Analyser:** analisador básico, que utiliza o Whitespace Tokenizer, que separa os tokens de um texto considerando um espaço em branco como separador;
- **Stop Analyser:** analisador que permite que configuremos `stop words` , que nada mais são do que palavras que o analisador vai usar como separadores para os tokens a serem analisados;
- **Keyword Analyser:** analisador especial que classifica todo o dado recebido no campo como um único token em vez de tentar quebrar o dado em tokens. Útil quando temos campos cujos dados possuem uma atomicidade longa, como longos IDs, por exemplo;
- **Pattern Analyser:** análogo ao *Stop Analyser*, que permite que configuremos `stop words` . Este também permite que configuremos como o dado será separado em tokens, porém, neste caso, nos permitindo

configurar expressões regulares para esse fim;

- **Language Analysers:** por default, o Elasticsearch processa textos em língua inglesa. Com essa série de analisadores, podemos analisar textos em diferentes línguas. Atualmente são suportadas línguas como árabe, armênio, basco, português brasileiro, português de Portugal, búlgaro, catalão, checo, holandês, francês, italiano, persa, norueguês, romeno e muitas outras!

Mas de que forma podemos utilizar esses diferentes analisadores em nossas consultas e mapeamentos? Vejamos agora alguns exemplos de manipulação de analisadores.

> **Importante**
>
> Não é possível alterar o analisador de um campo, uma vez que ele já tenha sofrido indexações! Apenas a adição de novos analisadores customizados ao índice e o *setting* de analisadores diferentes do default para novos campos são permitidos na API de alterações do Elasticsearch.

Na nossa construção anterior da nossa massa de dados de produtos e categorias, nossos dados estão em português. Entretanto, o analisador default do Elasticsearch é na língua inglesa, ou seja, não é o mais adequado para os nossos dados. Para isso, faremos a seguinte sequência de passos:

1. Remover os mapeamentos de produtos e categorias (cuidado com isso em produção! Você perderá os seus dados!);
2. Executar o comando `_close` que fecha o índice;

3. Modificar o `settings` do índice para criar um analisador *custom* de português. Para isso, vamos usar o próprio exemplo do site da Elastic (https://www.elastic.co/guide/en/elasticsearch/reference/2.0/analysis-lang-analyzer.html#portuguese-analyzer). Esses passos de criação do analisador *custom* na verdade não são necessários, pois poderíamos perfeitamente utilizar o já pronto, mas vamos usar este formato apenas para ilustrar o recurso de customização;

4. Reabrir o índice com o comando `_open` ;

5. Recriar os mapeamentos, referenciando o novo analisador que acabamos de criar;

6. Reindexar os documentos com o novo analisador.

Com os passos definidos, mãos a obra!

Vamos começar removendo os mapeamentos de produtos e categorias. Para isso, basta executar os comandos:

```
curl -XDELETE 'http://localhost:9200/loja/produto'
curl -XDELETE 'http://localhost:9200/loja/categoria'
```

> Lembre-se de que, no Elasticsearch 2.x, não é possível remover os mapeamentos de documentos! Se você estiver utilizando o Elasticsearch 2.x, a alternativa será recriar o índice.

A seguir, vamos executar o comando `_close` para fechar o índice. Atente-se que esse comando deixa o índice indisponível enquanto ele não for reaberto, portanto, muito cuidado ao utilizar esse procedimento em ambiente produtivo. Podemos ver o

comando a seguir:

```
curl -X POST 'http://localhost:9200/loja/_close'
```

Vamos agora modificar o `settings`. Criaremos um novo analisador, chamado de `meu_portugues`. Perceba que, na estrutura, definimos e/ou reutilizamos diferentes componentes que compõem um analisador, como *character filters* e *tokenizers*:

```
curl -X PUT 'http://localhost:9200/loja/_settings' -d '{
"analysis": {
      "filter": {
        "meu_portugues_stop": {
          "type":       "stop",
          "stopwords":  "_portuguese_"
        },
        "meu_portugues_stemmer": {
          "type":       "stemmer",
          "language":   "light_portuguese"
        }
      },
      "analyzer": {
        "meu_portugues": {
          "tokenizer":  "standard",
          "filter": [
            "lowercase",
            "meu_portugues_stop",
            "meu_portugues_stemmer"
          ]
        }
      }
    }
}'
```

Agora que criamos o nosso novo analisador, vamos reabrir o índice com o comando `_open`:

```
curl -X POST 'http://localhost:9200/loja/_open'
```

Por fim, recriaremos os mapeamentos. Repare que incluímos o campo `analyser`, onde indicamos que os campos de texto usem o nosso recém-criado analisador. Se fôssemos simplesmente utilizar o

analisador pronto do Elasticsearch, bastaria fazer essa modificação, colocando na propriedade do analisador o valor `portuguese` :

```
curl -XPUT 'localhost:9200/loja/categoria/_mapping' -d '{
  "categoria" : {
    "properties" : {
      "nome" : {"type" : "string",
                "analyser" : "meu_portugues"
                }
    }
} }'

curl -XPUT 'localhost:9200/loja/produto/_mapping' -d '{
  "produto" : {
  "_parent" : { "type" : "categoria" },
    "properties" : {
      "sku" : {"type" : "integer"},
      "nome" : {"type" : "string",
                "analyser" : "meu_portugues"},
      "descricao" : {"type" : "string",
                "analyser" : "meu_portugues"},
      "marca" : {"type" : "string",
                "analyser" : "meu_portugues"},
      "estoque" : {"type" : "integer"}
    }
} }'
```

> Na versão 2.x do Elasticsearch, a opção de se definir diferentes analisadores para um mesmo campo em tempo de indexação e/ou busca foi removida, como pode ser visto no link: https://github.com/elastic/elasticsearch/issues/9279. Para o Elasticsearch 2.x, temos a alternativa de criar templates dinâmicos, que veremos na próxima seção.

Com os mapeamentos criados, basta reexecutar a carga dos dados. Não é necessário fazer nenhuma alteração no código anterior, por isso vamos omitir essa parte, bastando obter os comandos do arquivo `comandoselasticsearch.txt` .

Vamos agora testar uma consulta para ver se está tudo bem com o nosso índice. Executaremos novamente a query seguinte, da nossa seção que demonstrou a query `has_child` :

```
curl -XGET 'localhost:9200/loja/_search?pretty=true&routing=Eletro'
 -d '{
  "query" : {
    "has_child" : {
        "type" : "produto",
        "query": {
          "query_string" : {
              "query": "marca:marcaA"
          }
        }
     }
  }
}'
```

Veremos que a busca retornou resultados como anteriormente, provando que toda a nossa configuração foi um sucesso.

## 5.6 TEMPLATES DINÂMICOS

Conforme vimos na observação da seção anterior, no Elasticsearch 2.x não temos mais a opção de definir diferentes analisadores para um mesmo campo. No Elasticsearch 2.x, uma alternativa para utilizar diferentes linguagens é o uso de templates dinâmicos.

Com essa alternativa, podemos definir regras para todos os campos a serem criados dentro de um determinado índice, a partir do seu tipo ou de parte do seu nome. Vamos usar este recurso agora para definir o nosso analisador em português para todos os campos `string` do nosso índice `loja` .

Para isso, partindo do princípio que o leitor esteja com o índice `loja` recém-criado (ou seja, vazio), execute o comando a seguir para criar os templates dinâmicos, juntamente com os próprios

mapeamentos dos tipos. Repare que, neste cenário, não utilizamos o campo `_analyser` :

```
curl -XPUT 'http://localhost:9200/loja' -d '{
    "mappings": {
        "produto": {
            "_parent" : { "type" : "categoria" },
            "properties" : {
            "sku" : {"type" : "integer"},
            "nome" : {"type" : "string"},
            "descricao" : {"type" : "string"},
            "marca" : {"type" : "string"},
            "estoque" : {"type" : "integer"}
            },
            "dynamic_templates": [
                { "pt": {
                      "match":               "*",
                      "match_mapping_type": "string",
                      "mapping": {
                          "type":           "string",
                          "analyzer":       "meu_portugues"
                      }
                }}
            ]
},
"categoria" : {
    "properties" : {
      "nome" : {"type" : "string"
                }
    },
    "dynamic_templates": [
                { "pt": {
                      "match":               "*",
                      "match_mapping_type": "string",
                      "mapping": {
                          "type":           "string",
                          "analyzer":       "meu_portugues"
                      }
                }}
            ]
}
}}'
```

Após executar esse comando, basta executarmos a criação do analisador, seguindo os passos da seção anterior, desde o fechamento do índice até o passo de reabertura.

Vamos agora testar o analisador. Para isso, basta reexecutar a carga dos dados. Não é necessário fazer nenhuma alteração no código anterior, por isso vamos omitir essa parte, bastando obter os comandos do arquivo `comandoselasticsearch.txt` .

Vamos agora testar uma consulta para ver se está tudo bem com o nosso índice. Vamos executar novamente a query adiante da nossa seção que demonstrou a query `has_child` :

```
curl -XGET 'localhost:9200/loja/_search?pretty=true&routing=Eletro'
 -d '{
  "query" : {
    "has_child" : {
        "type" : "produto",
        "query": {
          "query_string" : {
              "query": "marca:marcaA"
          }
        }
     }
  }
}'
```

Veremos que a busca retornou resultados como anteriormente, provando que toda a nossa configuração foi um sucesso.

Agora que concluímos nosso estudo sobre o *core* dos analisadores do Elasticsearch, vamos aprender sobre mais alguns tipos interessantes de consultas que podemos realizar nele, utilizando nossa recém-adquirida massa de dados do Twitter.

## 5.7 OUTROS MODOS DE CONSULTA DO ELASTICSEARCH

### Consultas por termos comuns

Esse tipo de consulta fornece um mecanismo bastante poderoso que permite que possamos efetuar consultas por conjuntos de

termos que contém termos muito comuns, que tradicionalmente teriam pouco impacto na consulta, mas que podem ter impacto na precisão dos resultados a serem encontrados se desconsiderados.

Imaginemos uma consulta em um dado campo pela frase em inglês *"I don't like coca cola"*. No método tradicional de cálculo de score do Elasticsearch – consultar a seção *Score* do nosso capítulo *Dissecando a ELK – Elasticsearch* para maiores informações –, os termos *"I"* e *"dont"* teriam pouca relevância na consulta, devido a naturalmente serem palavras muito comuns da língua inglesa e, consequentemente, estarem presentes em muitos documentos.

Porém, neste caso, não considerar o termo *"dont"* nesta consulta causa um impacto dramático em sua precisão, pois isso é uma parte muito importante do sentido que se deseja consultar. Queremos consultar por pessoas que estão dizendo que não gostam de coca-cola, e o "dont" é precisamente o termo chave que nos permite chegar nesses resultados.

Pensando nesse problema, a consulta por termos comuns foi criada. Nela, os termos são divididos em muito importantes (baixa frequência) e pouco importantes (alta frequência). Após essa divisão, o Elasticsearch efetua a consulta pelos termos do grupo muito importantes, extraindo uma lista de resultados. A seguir, o indexador efetua a consulta pelos termos do grupo pouco importantes, porém calculando o score *apenas* dos documentos que já foram encontrados na primeira consulta.

Dessa forma, os termos de baixa frequência ajudam a incrementar a qualidade dos resultados dos termos de alta frequência. Uma propriedade importante dessa consulta é a `cutoff_frequency`, que consiste de um valor de corte na frequência dos termos, de modo a definir em qual grupo eles deverão ser alocados.

Vamos então fazer uma consulta de exemplo na nossa base de *tweets*, pesquisando a frase que acabamos de ver como exemplo. Podemos realizar essa pesquisa pelo seguinte comando:

```
curl -XGET 'localhost:9200/twitter-*/_search?pretty' -d '{
  "query" : {
      "common": {
        "message": {
          "query": "I dont like coca cola",
          "cutoff_frequency": 0.001
        }
      }
 }
}'
```

Como podemos ver no fragmento a seguir, nossa consulta retornou com considerável relevância os documentos que queríamos encontrar para o que intencionávamos buscar – e alguém não gosta muito de Harry Potter.

```
{
  "took" : 27,
  "timed_out" : false,
  "_shards" : {
    "total" : 40,
    "successful" : 40,
    "failed" : 0
  },
  "hits" : {
    "total" : 63,
    "max_score" : 3.4473815,
    "hits" : [ {
      "_index" : "twitter-2015.11.11",
      "_type" : "logs",
      "_id" : "AVD0a_h5GEZKX2Upmv2k",
      "_score" : 3.4473815,
      "_source":{"@timestamp":"2015-11-11T02:42:07.000Z","message":
"i hate coca cola and i dont like cherrys but i dig cherry cola","
user":"slak666dozer","client":"<a href=\"http://twitter.com/downlo
ad/iphone\" rel=\"nofollow\">Twitter for iPhone</a>","retweeted":f
alse,"source":"http://twitter.com/slak666dozer/status/664271860004
052992","@version":"1"}
    }, {
      "_index" : "twitter-2015.11.06",
```

```
      "_type" : "logs",
      "_id" : "AVDe2Q_Qp3hS2xjYX0uM",
      "_score" : 3.1537242,
      "_source":{"@timestamp":"2015-11-06T22:09:38.000Z","message":
"@tara100cards @littleonepaige @amazon I DONT LIKE HARRY POTTER","
user":"AkiraArruda","client":"<a href=\"http://twitter.com/downloa
d/iphone\" rel=\"nofollow\">Twitter for iPhone</a>","retweeted":fa
lse,"source":"http://twitter.com/AkiraArruda/status/66275373443964
1090","@version":"1","in-reply-to":662628661728800768}
    }, {
      "_index" : "twitter-2015.11.02",
      "_type" : "logs",
      "_id" : "AVDJnHXoYhpRlW1kJcJm",
      "_score" : 2.6392646,
      "_source":{"@timestamp":"2015-11-02T19:11:25.000Z","message":
"@MegsAtTheDisco DEAR TWITTER I DONT TAKE GEAR, I MENT COCA-COLA",
"user":"caitlincc_","client":"<a href=\"http://twitter.com/downloa
d/iphone\" rel=\"nofollow\">Twitter for iPhone</a>","retweeted":fa
lse,"source":"http://twitter.com/caitlincc_/status/661259333108432
897","@version":"1","in-reply-to":660488677525729280}
    }
..... restante omitido ......
```


## Consultas more like this

Em consultas *"more like this"*, pedimos ao Elasticsearch que consulte para nós por mais documentos que sejam "parecidos" com documentos que já pesquisamos anteriormente, mesclando com outros textos para consulta se necessário. Por exemplo, se nós desejássemos encontrar mais documentos parecidos com o documento anterior que criticou Harry Potter, além do texto *"amazon"*, poderíamos fazer uma consulta como a seguinte:

```
curl -XGET 'localhost:9200/twitter-*/_search?pretty' -d '{
  "query" : {
      "more_like_this" : {
        "fields" : ["message"],
        "docs" : [
        {
            "_index" : "twitter-2015.11.06",
            "_type" : "logs",
            "_id" : "AVDe2Q_Qp3hS2xjYX0uM"
        }, {
        "_index" : "twitter-*",
```

```
        "_type" : "logs",
        "doc" : { "message" : "amazon" }
        }]
    }
 }
}'
```

Esse tipo de consulta pode ser muito útil em sistemas que quiséssemos implementar recursos de consulta mais aprimorados, como por exemplo, mecanismos de consulta em que o usuário pudesse pesquisar por mais documentos relacionados a um determinado documento de sua consulta antecessora.

## Consultas dismax

Em consultas do tipo *"dismax"*, temos a opção de realizar a união de diferentes subqueries. Documentos que são encontrados por mais de uma subquery têm o seu maior valor de score mantido para a montagem do resultado final. Tal consulta pode ser útil quando desejamos consultar documentos realizando a filtragem por múltiplos campos.

Opcionalmente, é possível utilizar a propriedade `tie_breaker` , onde um incremento no score pode ser adicionado para as situações em que o documento for encontrado em mais de um campo.

Vamos ver um exemplo desse tipo de consulta. Vamos consultar todos os tweets que contenham as palavras *"java"* e *"android"*, com um score adicional para os casos em que encontrarmos ambas:

```
curl -XGET 'localhost:9200/twitter-*/_search?pretty' -d '{
  "query" : {
      "dis_max" : {
        "tie_breaker" : 0.7,
        "queries" : [
            {
                "term" : { "message" : "java" }
            },
            {
```

```
                "term" : { "message" : "android" }
            }
        ]
    }
 }
}'
```

Como podemos ver no fragmento dos meus resultados com base na minha massa do Twitter, a consulta se comportou do modo esperado:


```
{
  "took" : 11,
  "timed_out" : false,
  "_shards" : {
    "total" : 40,
    "successful" : 40,
    "failed" : 0
  },
  "hits" : {
    "total" : 10232,
    "max_score" : 3.8019264,
    "hits" : [ {
      "_index" : "twitter-2015.11.02",
      "_type" : "logs",
      "_id" : "AVDKQYHJj1pM6JX9MiHR",
      "_score" : 3.8019264,
      "_source":{"@timestamp":"2015-11-02T22:11:41.000Z","message":
"@ProgramizeMe Java → Android","user":"Could_Blood","client":"<a h
ref=\"http://twitter.com/download/android\" rel=\"nofollow\">Twitt
er for Android</a>","retweeted":false,"source":"http://twitter.com
/Could_Blood/status/661304698209763328","@version":"1","in-reply-t
o":661303853543899136}
    }, {
      "_index" : "twitter-2015.11.02",
      "_type" : "logs",
      "_id" : "AVDJo-50YhpRlW1kJc1S",
      "_score" : 2.793043,
      "_source":{"@timestamp":"2015-11-02T19:19:34.000Z","message":
"Java Java Java Java Java Java Java Java Java Java Java Java Java
Java Java Java","user":"lilshake_mcjuke","client":"<a href=\"http:
//twitter.com/download/iphone\" rel=\"nofollow\">Twitter for iPhon
e</a>","retweeted":false,"source":"http://twitter.com/lilshake_mcj
uke/status/661261383363858432","@version":"1"}
    }, {
      "_index" : "twitter-2015.11.11",
```

```
      "_type" : "logs",
      "_id" : "AVDz5IglGEZKX2Upml5y",
      "_score" : 2.7333467,
      "_source":{"@timestamp":"2015-11-11T00:14:11.000Z","message":
"java","user":"i_t_c_account","client":"<a href=\"http://twittbot.
net/\" rel=\"nofollow\">twittbot.net</a>","retweeted":false,"sourc
e":"http://twitter.com/i_t_c_account/status/664234629705003008","@
version":"1"}
    },
..... restante omitido ......
```

## 5.8 FILTROS E CACHEAMENTO DE QUERIES

Conforme vimos no capítulo *Dissecando a ELK – Elasticsearch*, dentro do Elasticsearch temos um tipo de query chamada *filtered queries*, que nos trazem duas grandes vantagens:

- A filtered query não precisará executar os cálculos de score, bastando para ela saber quais documentos satisfazem a condição utilizada dentro do filtro;
- Filtered queries podem ter os seus filtros cacheados quando usados em conjunto, obtendo um grande incremento de performance.

Assim sendo, vamos montar agora alguns exemplos de filtered queries, além de utilizar o mecanismo de cache.

Quando montamos uma filtered query, temos 2 seções a serem especificadas: a `query` que realiza a consulta propriamente dita nos índices, e a `filter` que aplica uma filtragem nos resultados, ou seja, sem empregar o cálculo de score. Por exemplo, se quisermos pesquisar na nossa massa do Twitter por tweets que contenham o texto *"I love Elasticsearch!"* e que não tenham sido retweetados, fazemos a seguinte query:

```
curl -XGET 'localhost:9200/twitter-*/_search?pretty' -d '{
  "query" : {
```

```
  "filtered": {
    "query": {
      "match": { "message": "I love elasticsearch!" }
    },
    "filter": {
      "bool" : {
        "must" : {
            "term" : { "retweeted" : false }
         }
        }
    }
   }
  }
}'
```

Pelo que podemos ver do nosso fragmento, a consulta foi um sucesso, com alguns desvios devido ao uso de palavras muito comuns como *"love"*, é claro, mas retornando alguns resultados interessantes:

```
..... restante omitido ......


"hits" : {
    "total" : 12396,
    "max_score" : 2.018886,
    "hits" : [ {
      "_index" : "twitter-2015.11.06",
      "_type" : "logs",
      "_id" : "AVDexPdkp3hS2xjYXzDx",
      "_score" : 2.018886,
      "_source":{"@timestamp":"2015-11-06T21:47:41.000Z","message":
"I love how an ElasticSearch cluster on Docker just works. Almost
a little too magical, but I'll take it for a dev environment","use
r":"DavidAntaramian","client":"<a href=\"http://tapbots.com/softwa
re/tweetbot/mac\" rel=\"nofollow\">Tweetbot for Mac</a>","retweete
d":false,"source":"http://twitter.com/DavidAntaramian/status/66274
8210163765248","@version":"1"}
    }, {
      "_index" : "twitter-2015.11.04",
      "_type" : "logs",
      "_id" : "AVDUjNF1pX7KXgHP6sO4",
      "_score" : 1.5818876,
      "_source":{"@timestamp":"2015-11-04T22:10:08.000Z","message":
"I love amazon(:","user":"megan_roberto","client":"<a href=\"http:
//twitter.com/download/iphone\" rel=\"nofollow\">Twitter for iPhon
e</a>","retweeted":false,"source":"http://twitter.com/megan_robert
```

o/status/662029087263891457","@version":"1"}
    }, {
      "_index" : "twitter-2015.11.01",
      "_type" : "logs",
      "_id" : "AVDDcI64WTWL-vqP7ypu",
      "_score" : 1.5416412,
      "_source":{"@timestamp":"2015-11-01T14:25:44.000Z","message":
"I love Amazon","user":"randy_sparano","client":"<a href=\"http://
twitter.com/download/iphone\" rel=\"nofollow\">Twitter for iPhone<
/a>","retweeted":false,"source":"http://twitter.com/randy_sparano/
status/660825052406013953","@version":"1"}
    }, {
      "_index" : "twitter-2015.11.04",
      "_type" : "logs",
      "_id" : "AVDUt5MkpX7KXgHP6v3Y",
      "_score" : 1.5068763,
      "_source":{"@timestamp":"2015-11-04T22:56:51.000Z","message":
"I freaking love Amazon #reasonsiambroke","user":"thatladylyss","c
lient":"<a href=\"http://twitter.com/download/iphone\" rel=\"nofol
low\">Twitter for iPhone</a>","retweeted":false,"source":"http://t
witter.com/thatladylyss/status/662040840404103168","@version":"1"}
    }, {
      "_index" : "twitter-2015.11.10",
      "_type" : "logs",
      "_id" : "AVDzuuZJGEZKX2Upmijn",
      "_score" : 1.4295988,
      "_source":{"@timestamp":"2015-11-10T23:28:43.000Z","message":
"I love coca cola","user":"GeorgiaBernadi3","client":"<a href=\"ht
tps://mobile.twitter.com\" rel=\"nofollow\">Mobile Web (M5)</a>","
retweeted":false,"source":"http://twitter.com/GeorgiaBernadi3/stat
us/664223187383795712","@version":"1"}
    },

```
..... restante omitido ......
```

E onde está o cacheamento? Ele já é feito automaticamente pelo Elasticsearch, pois se tivéssemos mais de uma condição de filtro a aplicar na nossa query, os dados dos filtros executados anteriormente já estariam na memória.

## 5.9 CONCLUSÃO

E assim concluímos o nosso tour pelos recursos mais avançados

do Elasticsearch. Espero ter conseguido passar para você, leitor, uma boa ideia do quão poderoso é o Elastisearch, demonstrando como podemos utilizá-lo em diversos outros tipos de soluções, além de um simples analisador de logs.

Convido o leitor agora a seguir comigo para o próximo capítulo, onde vamos aprender como administrar um cluster de nós do Elasticsearch.

CAPÍTULO 6

# ADMINISTRANDO UM CLUSTER ELASTICSEARCH

Agora que já aprendemos como utilizar nosso Elasticsearch e ferramentas relacionadas, vamos aprender como administrar um cluster, tratando as principais questões relacionadas ao assunto.

## 6.1 MONTANDO O CLUSTER

A forma mais rápida e fácil de montar um cluster de Elasticsearch é muito simples: basta abrir várias janelas de terminal e iniciar o executável do Elasticsearch em cada uma delas! É isso mesmo, simples assim, basta iniciar o executável em várias janelas de terminal.

A figura a seguir mostra a subida de um segundo nó de Elasticsearch, onde podemos ver no console informações que demonstram o estabelecimento de um cluster:

```
Downloads — elasticsearch-2.1.0/bin/elasticsearch — elasticsearch-2.1.0/bin/e...
[2015-11-25 21:57:16,252][INFO ][plugins                  ] [Firearm] loaded [],
 sites []
[2015-11-25 21:57:16,270][INFO ][env                      ] [Firearm] using [1]
data paths, mounts [[/ (/dev/disk1)]], net usable_space [51.4gb], net total_spac
e [157.3gb], spins? [unknown], types [hfs]
[2015-11-25 21:57:17,342][INFO ][node                     ] [Firearm] initialize
d
[2015-11-25 21:57:17,342][INFO ][node                     ] [Firearm] starting .
..
[2015-11-25 21:57:17,415][INFO ][transport                ] [Firearm] publish_ad
dress {127.0.0.1:9301}, bound_addresses {127.0.0.1:9301}, {[fe80::1]:9301}, {[::
1]:9301}
[2015-11-25 21:57:17,421][INFO ][discovery                ] [Firearm] elasticsea
rch/kp_MUc4ySaCdIpBomZL0Pw
[2015-11-25 21:57:20,481][INFO ][cluster.service          ] [Firearm] detected_m
aster {Demogorge the God-Eater}{lVYKJCo2SFGWTXqdso7M2g}{127.0.0.1}{127.0.0.1:930
0}, added {{Demogorge the God-Eater}{lVYKJCo2SFGWTXqdso7M2g}{127.0.0.1}{127.0.0.
1:9300},}, reason: zen-disco-receive(from master [{Demogorge the God-Eater}{lVYK
JCo2SFGWTXqdso7M2g}{127.0.0.1}{127.0.0.1:9300}])
[2015-11-25 21:57:20,514][INFO ][http                     ] [Firearm] publish_ad
dress {127.0.0.1:9201}, bound_addresses {127.0.0.1:9201}, {[fe80::1]:9201}, {[::
1]:9201}
[2015-11-25 21:57:20,514][INFO ][node                     ] [Firearm] started
```

Figura 6.1: Console de start de um segundo nó de Elasticsearch

Mas como essa *mágica* é feita? É o que vamos descobrir a seguir.

## 6.2 DESCOBERTA DE NÓS (DISCOVERY)

Por default, o Elasticsearch utiliza um mecanismo de descoberta de nós chamado *zen*, que por meio de um mecanismo de multicast da rede realiza uma varredura atrás de outros nós de Elasticsearch na rede. O nó responsável pelo gerenciamento do cluster, que chamamos de nó mestre, comumente é estabelecido como sendo o primeiro nó a ser inicializado dentro da rede. Entretanto, em caso de baixa desse nó, o Elasticsearch automaticamente seleciona outro para assumir esse papel.

Na última figura que gerei a partir de um cluster de 2 nós que estabeleci, podemos constatar uma mensagem no log que indica de que se trata do segundo nó, estabelecendo um canal de comunicação com o primeiro nó que foi levantado, como podemos ver a seguir:

```
2015-11-25 21:57:20,481][INFO ][cluster.service          ] [Firear
m] detected_master {Demogorge the God-Eater}{lVYKJCo2SFGWTXqdso7M2
g}{127.0.0.1}{127.0.0.1:9300}, added {{Demogorge the God-Eater}{lV
YKJCo2SFGWTXqdso7M2g}{127.0.0.1}{127.0.0.1:9300},}, reason: zen-di
sco-receive(from master [{Demogorge the God-Eater}{lVYKJCo2SFGWTXq
dso7M2g}{127.0.0.1}{127.0.0.1:9300}])
```

Opcionalmente, também podemos configurar a descoberta de nós para ser feita via unicast. A eleição do nó mestre também pode ser modificada ao alterarmos o arquivo `elasticsearch.yml` , dentro da pasta `config` da instalação do Elasticsearch. Lá podemos configurar que um nó nunca venha a ser elegível como mestre, se conveniente. Nas próximas seções, vamos aprender que outras configurações são possíveis de se fazer através desse arquivo.

Se subirmos o nosso exercício do uso do ELK deste livro, veremos que ele funcionará com o cluster sem problemas. Porém, não estaremos aproveitando todo o potencial do nosso cluster: as chamadas do nosso Logstash não vão balancear pelos nós do cluster, assim como também as chamadas do Kibana não sofrerão balanceamento. Mas de que modo podemos corrigir essa questão? Por meio dos passos que veremos a seguir.

## 6.3 CONFIGURANDO O CLUSTER: CONFIGURAÇÕES NO LOGSTASH

Para configurar o Logstash para usar o nosso cluster, é muito simples, basta adicionarmos o nó dentro da propriedade `hosts` do plugin de output do Elasticsearch de nossos fluxos. Se estivermos usando as configurações default, a porta do nosso segundo nó é a 9201, sendo assim, basta fazermos uma alteração como a do exemplo:

```
...restante omitido
```

```
    elasticsearch {
          hosts => [ "localhost:9200","localhost:9201" ]

...restante omitido
```

Pronto! Agora já podemos usar todo o poder do Elasticsearch em nossos *streams* do Logstash!

## 6.4 CONFIGURANDO O CLUSTER: CONFIGURAÇÕES NO KIBANA

Do lado do Kibana, a configuração é ligeiramente mais complicada, mas nada fora do comum. A recomendação oficial da Elastic é subir um novo nó de Elasticsearch no nosso cluster, configurando o mesmo como um nó apenas de busca e não elegível a mestre do cluster.

Todas essas configurações são feitas pelas propriedades `node.master` , `node.data` e `cluster.name` , que veremos com mais detalhes na próxima seção. Uma vez tendo o nosso nó devidamente configurado, basta configurarmos a propriedade `elasticsearch_url` do arquivo `kibana.yml` da instalação do nosso Kibana, apontando para o ip e porta do nó que subimos.

Para maiores informações sobre este e outros assuntos relacionados ao uso do Kibana em ambiente produtivo, consulte https://www.elastic.co/guide/en/kibana/current/production.html#load-balancing.

## 6.5 O ARQUIVO DE CONFIGURAÇÃO PRINCIPAL DO ELASTICSEARCH

Conforme acabamos de ver, o arquivo principal de configuração do Elasticsearch se chama `elasticsearch.yml` , dentro da pasta `config` da sua instalação. Dentro desse arquivo, também existe o

arquivo de configuração `logging.yml` , porém ele trata apenas das configurações de log do cluster, como os log levels, onde os arquivos de log serão armazenados etc. Portanto, devido a ele ter pouca relação com as configurações de administração do cluster, não o abordaremos neste livro.

Dentro do arquivo `elasticsearch.yml` , podemos fazer as seguintes configurações:

- `cluster.name` : identificador de um cluster de Elasticsearch, essa configuração é útil em cenários que temos múltiplos clusters coexistindo em uma única rede, pois os nós realizaram a descoberta apenas dos nós que compartilharem o mesmo `cluster.name` ;
- `node.name` : identificador que define de forma fixa o nome do nó de Elasticsearch que vamos subir com a nossa instalação. Obviamente, se desejarmos configurar esta propriedade, teremos de ter uma instalação separada para cada nó;
- `node.data` : propriedade que define que o nó de Elasticsearch desta instalação não reterá dados, ou seja, será um nó usado apenas para efetuar consultas no cluster;
- `path.data` : propriedade que altera o caminho onde o ELasticsearch vai armazenar os dados indexados;
- `path.logs` : propriedade que altera o caminho onde o Elasticsearch vai armazenar os logs de execução;
- `path.repo` : propriedade que define a localização no *file system* dos repositórios de snapshots do Elasticsearch – maiores informações sobre o que são snapshots podem ser encontradas na seção *Backup &*

*restore*, mais adiante;

- `bootstrap.mlockall` : propriedade que define se desejamos utilizar o recurso `mlockall` do sistema Linux/Unix, ou VirtualLock no Windows. Esse recurso permite que o Elasticsearch bloqueie o espaço reservado para ele na memória RAM, evitando perda de memória;

- `network.host` : propriedade que estabelece um ip fixo de bind do Elasticsearch para as suas requisições;

- `http.port` : propriedade que modifica a porta default do Elasticsearch, onde desejamos que este responda as requisições de suas APIs REST;

- `gateway.recover_after_nodes` : propriedade que define o processo de recuperação de índices, que ocorre toda vez que um nó de Elasticsearch é iniciado, seja executado somente após o estabelecimento de um cluster de X nós, onde X é o valor definido nessa propriedade;

- `discovery.zen.ping.unicast.hosts` : propriedade em que definimos um *array* de endereços IPs de hosts, que o Elasticsearch vai utilizar para efetuar o discovery de nós na sua inicialização;

- `discovery.zen.minimum_master_nodes` : propriedade que define o número mínimo de nós que precisam estar ativos no cluster para que uma nova eleição de nó master seja feita, evitando assim um problema chamado *split brain*, que falaremos a seguir;

- `node.max_local_storage_nodes` : propriedade que impossibilita que múltiplos nós sejam inicializados em

uma mesma máquina, como demonstrei no início deste capítulo. Tal propriedade pode ser interessante em um ambiente produtivo virtualizado, onde tipicamente teríamos uma máquina virtual por nó criado;

- `action.destructive_requires_name` : propriedade que obriga que sejam especificados os nomes completos dos índices no comando de deleção de índices. Se tivermos essa opção habilitada, não poderemos realizar comandos como deletar vários índices com uma única chamada pelo uso de wildcards – que é permitido por default –, trazendo assim uma segurança adicional para a ação;

Mas de que se trata esse problema do *split-brain*? É sobre isso que falaremos agora.

## 6.6 RESOLVENDO O SPLIT-BRAIN DE UM CLUSTER ELASTICSEARCH

Imaginemos um cluster de 2 nós, na seguinte situação:

1. Os dois nós estavam operantes trocando informações, com o nó 1 operando como master do cluster;

2. Por alguma razão, a comunicação entre os dois é perdida, porém sem a queda de qualquer dos nós, sendo causada por outras razões, como por exemplo, a queda de um componente de rede;

3. O nó 1, que já era master, não sofrerá qualquer modificação. O nó 2, porém, que não terá como saber se o nó 1 está de fato inoperante ou não, assumirá também o papel de master, gerando então a situação em que os 2 nós assumirão o papel de mestre!

Esse é o problema que chamamos de *split-brain*. Essa situação é ruim, pois nesse cenário, teremos dados sendo indexados nos dois nós, que não terão nenhum tipo de associação entre si para permitir uma consulta uniforme, independente do nó usado nas pesquisas, mesmo depois que a comunicação for restabelecida.

É nesse cenário que a propriedade que falamos anteriormente entra em ação. Com a propriedade `discovery.zen.minimum_master_nodes` , configuramos que uma nova eleição de nós deve ser feita apenas se um mínimo de nós ainda estiver estabelecido e o nó master perder a comunicação com o grupo de nós, devido a algum problema de rede ou queda do próprio nó. Desse modo, os nós isolados não se autoestabelecem como cluster, evitando assim o problema do split-brain.

## 6.7 TUNNING

Agora que já vimos as principais configurações que podemos realizar nos arquivos de configuração do Elasticsearch, vamos aprender algumas possíveis configurações que podemos realizar para *tunar* o nosso Elasticsearch.

Todas as configurações que veremos a seguir se tratam de configurações no nível dos índices, feitas por meio das APIs REST do Elasticsearch.

### Frequência de refresh de índices

Dentro do Elasticsearch, existe um processo responsável por varrer a estrutura do índice, a fim de verificar a inclusão e/ou alteração nos mapeamentos dos documentos. Esse processo é muito importante para que possamos ter a estrutura atualizada o mais rápido possível, porém ela torna as consultas e indexações mais lentas, visto que o Elasticsearch é obrigado a disponibilizar recursos

de sua infraestrutura para essa operação.

Em cenários em que temos uma estrutura de documentos já formada, que não vão sofrer modificações, e temos uma quantidade muito grande de indexações e/ou consultas, pode ser interessante aumentar a frequência de refresh, ou até mesmo desabilitá-la, com o objetivo de liberarmos recursos do Elasticsearch para essas operações.

Realizar essa alteração é muito simples, basta executar um comando como o seguinte, onde estamos desabilitando o refresh para o nosso índice `loja`:

```
curl -XPUT 'localhost:9200/loja/_settings' -d '{
    "index" : {
        "refresh_interval" : "-1"
    } }'
```

**Importante**

Não esquecer que, se for necessário voltar a alterar as estruturas de documentos do índice, essa configuração deve ser reajustada!

## Alta disponibilidade (réplicas)

Quando temos um cluster, cada documento armazenado em cada índice do Elasticsearch possui a sua disponibilidade controlada através do uso de réplicas, onde cópias do documento são armazenadas em outros nós do cluster. Desse modo, se perdermos um nó, ainda teremos o dado disponível por meio das outras cópias.

Obviamente, esse procedimento de geração de réplicas tem o seu custo, que pode ser agravado quando temos cenários em que vários

milhões de documentos devem ser replicados pelo cluster.

Dependendo do cenário de criticidade das informações, podemos controlar essa quantidade de réplicas para um valor mínimo, de modo que o cluster gaste menos recursos com a geração das réplicas. Para modificar, por exemplo, que os documentos do nosso índice `loja` gerem apenas 1 réplica em todo o cluster, basta executarmos um comando como:

```
curl -XPUT 'localhost:9200/loja/_settings' -d '
{
    "index" : {
        "number_of_replicas" : 1
    }
}'
```

## 6.8 BACKUP & RESTORE

Dentro das ações de administração, algumas das operações mais comuns consistem no backup e restore, que nada mais são do que efetuar cópias de segurança, além de efetuar recuperações dessas cópias quando necessário. No Elasticsearch, isso é feito pela Elasticsearch Time Machine.

Para criar um backup, devemos criar primeiro um repositório de snapshots, que consistem de agrupamentos onde as cópias – chamadas *snapshots* – são armazenadas. Para criar um repositório apontando para uma pasta do nosso file system, basta executar um comando como o seguinte:

```
curl -XPUT 'localhost:9200/_snapshot/elasticsearch_backups' -d '{
  "type": "fs",
  "settings": {
    "location": "/Users/alexandrelourenco/elasticsearch_backups"
  }
}'
```

**Importante**

Antes de executar o comando anterior, é preciso configurar a propriedade `path.repo` dentro do arquivo `elasticsearch.yml`!

No nosso exemplo, estamos criando os backups no nosso próprio file system, porém, o Elasticsearch também possui opções prontas de criação de repositórios de snapshots em estruturas HDFS (*Hadoop Distributed File System*), e até mesmo nas soluções em nuvem da AWS e Azure. Após criarmos o repositório, podemos criar um backup pelo comando:

```
curl -XPUT 'localhost:9200/_snapshot/elasticsearch_backups/backup1'
```

Ao executarmos esse comando, recebemos uma resposta assíncrona, na qual o Elasticsearch apenas nos retorna um OK de que vai realizar o nosso backup, passando a realizá-lo em background. Se desejássemos executar o comando de modo que este só retorne após o término do backup, basta chamarmos o comando com a flag `wait_for_completion`, como podemos ver a seguir:

```
curl -XPUT 'localhost:9200/_snapshot/elasticsearch_backups/backup2
?wait_for_completion=true&pretty'
```

Esse comando produzirá uma resposta análoga como a seguinte:

```
{
  "snapshot" : {
    "snapshot" : "backup2",
    "version_id" : 2000099,
    "version" : "2.0.0",
    "indices" : [ "twitter-2015.11.19" ],
```

```
    "state" : "SUCCESS",
    "start_time" : "2015-11-20T00:01:44.767Z",
    "start_time_in_millis" : 1447977704767,
    "end_time" : "2015-11-20T00:01:44.818Z",
    "end_time_in_millis" : 1447977704818,
    "duration_in_millis" : 51,
    "failures" : [ ],
    "shards" : {
      "total" : 5,
      "failed" : 0,
      "successful" : 5
    }
  }
}
```

E produzirá, dentro da pasta que definimos como repositório de snapshots, uma estrutura de file system como:

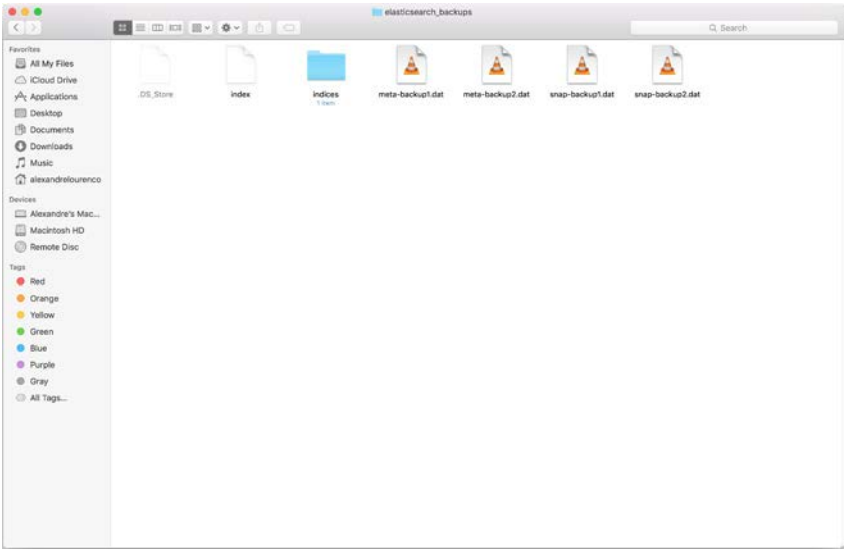



Figura 6.2: Pasta de backups do Elasticsearch

Mas e quanto ao restore? Primeiro, vamos listar todos os backups contidos no repositório, por meio do comando:

```
curl -XGET 'localhost:9200/_snapshot/elasticsearch_backups/_all?pr
etty'
```

Esse comando produzirá um resultado análogo ao seguinte:

```
{
  "snapshots" : [ {
    "snapshot" : "backup2",
    "version_id" : 2000099,
    "version" : "2.0.0",
    "indices" : [ "twitter-2015.11.19" ],
    "state" : "SUCCESS",
    "start_time" : "2015-11-20T00:01:44.767Z",
    "start_time_in_millis" : 1447977704767,
    "end_time" : "2015-11-20T00:01:44.818Z",
    "end_time_in_millis" : 1447977704818,
    "duration_in_millis" : 51,
    "failures" : [ ],
    "shards" : {
      "total" : 5,
      "failed" : 0,
      "successful" : 5
    }
  }, {
    "snapshot" : "backup1",
    "version_id" : 2000099,
    "version" : "2.0.0",
    "indices" : [ "twitter-2015.11.19" ],
    "state" : "SUCCESS",
    "start_time" : "2015-11-20T00:03:56.759Z",
    "start_time_in_millis" : 1447977836759,
    "end_time" : "2015-11-20T00:03:56.789Z",
    "end_time_in_millis" : 1447977836789,
    "duration_in_millis" : 30,
    "failures" : [ ],
    "shards" : {
      "total" : 5,
      "failed" : 0,
      "successful" : 5
    }
  } ]
}
```

E finalmente, para restaurarmos o backup, por exemplo, o backup de nome `backup1`, basta executarmos um comando como:

```
curl -XPOST 'localhost:9200/_snapshot/elasticsearch_backups/backup
1/_restore'
```

Um ponto importante é que, se nosso backup contiver algum índice que já existe no Elasticsearch, o comando vai falhar. Também é possível recuperar apenas alguns índices do backup, como por exemplo, no comando a seguir, onde só recuperaríamos o índice `loja` do backup:

```
curl -XPOST 'localhost:9200/_snapshot/elasticsearch_backups/backup
1/_restore?pretty' -d '{
"indices": "loja"}'
```

E assim concluímos nosso aprendizado sobre backup e restore, duas importantes funcionalidades de qualquer administração.

## 6.9 MONITORAÇÃO DA SAÚDE DO CLUSTER COM O WATCHER

Outro ponto importante em uma administração é a monitoração da saúde do cluster, em que podemos configurar ações como o envio de um e-mail para a caixa dos administradores em caso de queda de algum nó do cluster, por exemplo. Para isso, usamos o plugin Watcher do Elasticsearch.

Vamos primeiro instalar o Watcher. Para isso, encerramos a execução do Elasticsearch e executamos os seguintes comandos de instalação, dentro da sua pasta de instalação:

```
bin/plugin install elasticsearch/license/latest
bin/plugin install elasticsearch/watcher/latest
```

A seguir, vamos testar se o Watcher subiu adequadamente. Para isso, executamos o seguinte comando:

```
curl -XGET 'http://localhost:9200/_watcher/stats?pretty'
```

Este que deve produzir um resultado como o seguinte:

```
{
  "watcher_state" : "started",
  "watch_count" : 0,
  "execution_thread_pool" : {
    "queue_size" : 0,
    "max_size" : 0
  },
  "manually_stopped" : false
}
```

Vamos agora construir um exemplo simples. No nosso exemplo, o Watcher enviará um e-mail via Gmail, com uma mensagem informando que o status do cluster necessita de atenção.

Dentro da configuração de um alerta do watcher, temos 3 seções distintas:

- **Trigger:** seção onde configuramos detalhes como a periodicidade em que as verificações do alerta devem ser feitas etc.
- **Conditions:** seção onde configuramos o que o alerta deve verificar como condição válida para disparo. No nosso exemplo, vamos configurar um alerta que disparará se o cluster apresentar um status de indisponibilidade (vermelho).
- **Actions:** seção onde configuramos as ações a serem tomadas caso o alerta seja disparado. No nosso exemplo, configuraremos o envio de um e-mail.

Assim sendo, para configurarmos o nosso exemplo, onde configuramos um watcher que operará de 10 em 10 segundos, enviando um e-mail em caso de problemas na saúde do cluster, com o seguinte comando:

```
curl -XPUT 'http://localhost:9200/_watcher/watch/cluster_health_wa
tch' -d '{
```

```
  "trigger" : {
    "schedule" : { "interval" : "10s" }
  },
  "input" : {
    "http" : {
      "request" : {
       "host" : "localhost",
       "port" : 9200,
       "path" : "/_cluster/health"
      }
    }
  },
  "condition" : {
    "compare" : {
      "ctx.payload.status" : { "eq" : "red" }
    }
  },
  "actions" : {
    "send_email" : {
      "email" : {
        "to" : "johndoe@gmail.com",
        "subject" : "Cluster Status Warning",
        "body" : "Cluster status is RED"
      }
    }
  }
}'
```

Para o envio do e-mail, é necessário configurar uma conta `smtp` no Elasticsearch. Para isso, vamos abrir o arquivo `elasticsearch.yml` e incluir as seguintes propriedades:

```
watcher.actions.email.service.account:
  work:
    profile: gmail
    email_defaults:
      from: <remetente>
    smtp:
      auth: true
      starttls.enable: true
      host: smtp.gmail.com
      port: 587
      user: <login da conta de envio>
      password: <senha da conta de envio>

`
```

Assim, temos configurado o nosso watcher. Também é possível construir outros tipos de watchers, até mesmo em cima de consultas de índices de soluções! Para maiores informações, sugiro ao leitor ler a documentação oficial da Elastic no seguinte endereço: https://www.elastic.co/guide/en/watcher/current/getting-started.html.

**Observação**

Se o leitor estiver seguindo a ordem do capítulo, verá que, quando configurarmos o Shield, começaremos a ver alguns erros de autenticação. A razão para isso é porque o watcher que acabamos de configurar está tentando acessar o Elasticsearch sem autenticação. Para fazer com que os erros parem de ocorrer, basta remover o watcher com o comando:

```
curl -u administrador -XDELETE 'http://localhost:9200/_watche
r/watch/cluster_health_watch'
```

## 6.10 EXPURGA COM O CURATOR

Outro ponto muito importante na administração é a expurga dos dados antigos. Se não tratarmos essa questão, teremos um crescimento desenfreado no uso do disco e um decremento na performance das nossas consultas por parte do nosso Elasticsearch, o que definitivamente não é o que desejamos. Para isso, vamos utilizar o Curator e configurar períodos de retenção para os nossos índices.

É possível usar facilmente o Curator para os nossos índices, porque tivemos o cuidado de construí-los de modo que, a cada novo dia, um novo índice seja criado dentro do Elasticsearch. Isso torna a limpeza incrivelmente simples, bastando excluir os índices mais antigos. Carregue sempre consigo essa prática quando for criar um índice novo!

O Curator consiste de um script feito em Python, que permite que configuremos *daemons* de limpeza dos índices do Elasticsearch. Para instalar o Curator, a forma mais simples é utilizando o comando `pip` do Python, conforme:

```
pip install elasticsearch-curator
```

Caso o leitor não queira ter o Python instalado em sua máquina – mas recomendo fortemente que o estude, está perdendo uma ótima linguagem de programação! –, outras formas de instalação podem ser encontradas em: https://www.elastic.co/guide/en/elasticsearch/client/curator/current/installation.html.

Agora que já temos o Curator instalado, vamos configurar um comando de expurga para os nossos índices do Twitter. Por exemplo, se quisermos que todos os índices que foram criados a mais de um dia sejam excluídos, basta rodar o comando:

```
curator delete indices --time-unit days --older-than 1 --timestrin
g '%Y.%m.%d' --prefix twitter-
```

Também é possível fazer diversos outros tipos de expurga, como por exemplo, expurgar índices de acordo com um determinado prefixo/sufixo, por períodos de data, e até mesmo hora, pelo

tamanho dos índices etc. Também é possível configurar nos nossos padrões de índice para que ele use horas, se o volume for alto o suficiente para isso. Maiores informações sobre as possibilidades de configurações do Curator podem ser encontradas em: https://www.elastic.co/guide/en/elasticsearch/client/curator/current/subcommand.html.

## 6.11 SEGURANÇA COM O SHIELD

Um último ponto que discutiremos é a segurança do nosso Elasticsearch. Em tudo o que usamos até o presente momento, nosso Elasticsearch tem permanecido aberto, ou seja, qualquer um pode efetuar chamadas ao nosso cluster, desde que conheça o endereço dos endpoints. Isso pode ser um problema, principalmente se temos dados sensíveis nele, que não podem ser acessados por qualquer um.

Pensando nisso, a Elastic lançou outro plugin chamado *Shield*, que implementa uma camada de segurança no Elasticsearch. Para instalá-lo é muito simples, basta fazermos analogamente ao que fizemos com o watcher, executando a seguinte série de comandos, com o ELasticsearch parado:

```
bin/plugin install elasticsearch/license/latest
bin/plugin install elasticsearch/shield/latest
```

Em seguida, vamos configurar um usuário administrador para o nosso uso do Elasticsearch. Para isso, vamos executar o seguinte comando, dentro da sua pasta de instalação:

```
bin/shield/esusers useradd administrador -r admin
```

Será solicitado que entremos com a senha do usuário e, em seguida, o cadastro estará concluído. Vamos agora inicializar o

nosso Elasticsearch e efetuar um teste.

Agora que estamos com o Elasticsearch inicializado, vamos testar chamar o comando que lista todos os índices da instalação:

```
curl 'localhost:9200/_cat/indices?v'
```

Ao executarmos o comando, porém, receberemos esta mensagem de erro:

```
{"error":{"root_cause":[{"type":"security_exception","reason":"mis
sing authentication token for REST request [/_cat/indices?v]","hea
der":{"WWW-Authenticate":"Basic realm=\"shield\""}}],"type":"secur
ity_exception","reason":"missing authentication token for REST req
uest [/_cat/indices?v]","header":{"WWW-Authenticate":"Basic realm=
\"shield\""}},"status":401}%
```

A razão para isso é muito simples: não passamos a identificação do usuário que deseja se conectar ao Elasticsearch, por isso nossa execução foi bloqueada. Vamos agora executar novamente o comando, porém desta vez passando o usuário (a senha será solicitada no ato da execução do comando):

```
curl -u administrador 'localhost:9200/_cat/indices?v'
```

Após a execução, poderemos ver que o comando retornou adequadamente a listagem de índices.

Mas e quanto ao Kibana e o Logstash? O leitor pode constatar no capítulo *Dissecando a ELK – Kibana* que falamos sobre o arquivo de configuração `kibana.yml` e do próprio Kibana, que existem lá duas propriedades – `kibana_elasticsearch_username` e `kibana_elasticsearch_password`. Estas devem ser usadas nesse cenário, sendo configuradas com o usuário e senha necessários para a conexão com o Elasticsearch.

No caso do Logstash, também é possível realizar essa

configuração por meio das propriedades de conexão do plugin. Deixo como desafio para o leitor configurar o Kibana e o Logstash para esse cenário de segurança.

## 6.12 CONCLUSÃO

E assim concluímos o nosso tour básico pela administração de um cluster de Elasticsearch. Como pudemos ver, a sua administração é bastante simples, consistindo basicamente de configurações feitas via REST API e arquivo de configuração, além de alguns plugins para facilitar a nossa vida.

Com os avanços cada vez maiores em trazer o time de desenvolvimento para junto do time de operações (DevOps), é natural que cada vez mais tenhamos administrações de ferramentas sendo o mais simples possíveis.

Convido agora você, leitor, para o último capítulo da nossa história, onde vamos falar um pouco de alguns cases bem interessantes de uso do Elasticsearch.

CAPÍTULO 7

# CONSIDERAÇÕES FINAIS

Bem-vindo, caro leitor, ao último capítulo da nossa jornada. Nele, vamos ver alguns dos cases mais interessantes de uso do Elasticsearch no mercado. Conhecê-los pode ser muito útil para ajudar a convencer o seu chefe a usar o Elasticsearch na sua empresa! Vamos a eles.

## 7.1 CASES DE MERCADO

### Globo.com

Vamos começar com um case nacional. Todos conhecem o portal globo.com, um dos maiores portais de internet do país, parte de um dos maiores grupos de telecomunicações da América Latina. Com um impressionante fluxo diário de 25 milhões de usuários, o portal possui uma enorme gama de conteúdo audiovisual a servir todos os dias.

Gerir a busca de todo esse conteúdo gera grandes desafios, devido ao enorme fluxo de usuários *versus* a grande quantidade de dados a serem buscados. No passado, o grupo utilizou outras soluções para atacar estes desafios, porém enfrentaram problemas principalmente com a performance e a acurácia das consultas dos usuários.

Após trocarem a sua solução de busca para o Elasticsearch, foi possível obter resultados impressionantes, com processamentos

diários de cerca de 180 queries por segundo, retornando resultados em até 100 ms.

Você pode conhecer mais detalhes do case no link: https://www.elastic.co/use-cases/globo.

## Docker

É isso mesmo, a mundialmente famosa Docker, cuja plataforma de virtualização de containers já é usada no mundo todo, também utiliza o Elasticsearch.

No case da Docker, o desafio era fornecer uma solução que permitisse que a enorme biblioteca de containers da Docker tivesse um mecanismo de consulta altamente performático, que permitisse que os usuários pudessem encontrar rapidamente os containers necessários para os seus ambientes.

Eles acabaram utilizando o Elasticsearch para essa tarefa, com ótimos resultados, além de trazer um ganho de desafogo para a infraestrutura da Docker, que pode deixar parte de sua infraestrutura para outra solução.

Você pode conhecer mais detalhes do case no link: https://www.elastic.co/use-cases/docker.

## Uber

Na Qcon São Francisco 2015, foi feita uma apresentação da própria Uber, onde pudemos conhecer mais da sua arquitetura, que processa uma enorme quantidade de informações por segundo. Com um modelo de máquina de estados, representando os diferentes estágios de um chamado (*buscando*, *esperando*, *em trânsito* etc.), que são processados por meio de uma solução de *streaming*.

Essa solução utiliza tecnologias como o Apache Kafka para messageria, e o Apache Samza para realizar pré-agregações dos dados. A solução utiliza 2 Elasticsearches: um voltado para o transacional ("quente") e outro voltado para relatórios ("frio").

Você pode conhecer mais detalhes do case no link a seguir, extraído da apresentação da Qcon São Francisco 2015: https://qconsf.com/system/files/presentation-slides/qconsf-2015-stream_processing_in_uber.pdf.

## GitHub

O famoso serviço de versionamento GitHub também é um usuário do Elasticsearch. No seu caso, o cenário teve dois objetivos:

- Fornecer um mecanismo de consulta altamente escalável – substituindo uma solução em Apache Solr – para os usuários do serviço;
- Montar um mecanismo de detecção de problemas na plataforma GitHub, indexando todos os logs e outros tipos de fontes de informações técnicas que são geradas pela plataforma.

Como nos outros casos, o case obteve bons resultados em ambos os objetivos, sendo outro bom exemplo de uso do Elasticsearch.

Você pode conhecer mais detalhes do case no link: https://www.elastic.co/use-cases/github.

## Netflix

O famoso serviço de *streaming* de vídeo Netflix também é outro consumidor do Elasticsearch. Com uma solução que mescla outras ferramentas de Big Data como o Hadoop, o Elasticsearch foi utilizado como parte de uma solução de processamento analítico,

detectando tendências, recomendações para os clientes, entre outros requisitos.

Esse case em especial é interessante de conhecer, pois ele não utiliza apenas o Elasticsearch, mas também outras ferramentas que vimos neste livro, como o Kibana.

Você pode conhecer mais detalhes do case no ótimo vídeo de uma apresentação da Netflix, no link abaixo:

https://www.elastic.co/videos/netflix-using-elasticsearch

### The Guardian

O famoso jornal inglês The Guardian é outro usuário do Elasticsearch. Para ser mais exato, o website theguardian.com, com 5 milhões de acessos por dia, é o terceiro maior site em língua inglesa do mundo.

Analogamente a solução da Globo.com, também nesta solução temos o Elasticsearch sendo usado como um poderoso mecanismo de busca de uma base de mais de 360 milhões de documentos, além de permitir a geração de relatórios que permitem análises como o impacto que uma determinada notícia teve na audiência do site.

Você pode conhecer mais detalhes do case no link: https://www.elastic.co/use-cases/guardian.

## 7.2 E AGORA, O QUE ESTUDAR?

Estamos chegando ao final da nossa jornada no mundo do Elasticsearch. Porém, isso não é tudo o que podemos fazer em um ecossistema Elasticsearch. Aconselho o leitor a entrar no site https://www.elastic.co, onde podemos ver várias novas ferramentas recém-lançadas em sua primeira versão, como:

- **Packetbeat:** ferramenta que permite que coletemos dados de pacotes de redes para o Elasticsearch;
- **Filebeat:** análogo ao Packetbeat, esta ferramenta permite que coletemos dados de arquivos do file system para o Elasticsearch. É verdade que podemos fazer isso também com o Logstash, porém esta alternativa é mais leve que um stream de Logstash, tornando-se uma boa opção de arquitetura termos vários processos de Filebeat alimentando um único processo de Logstash centralizado;
- **Topbeat:** completando a família *beat*, temos o Topbeat, que permite coletar dados da própria máquina, como RAM, processador etc. de um dado servidor para o Elasticsearch;
- **ES-Hadoop:** conector que permite que conectemos um cluster Elasticsearch com um cluster Apache Hadoop;
- Entre outras.

Aconselho a visitar o site para conhecer todas as novidades!

## 7.3 CONCLUSÃO

E assim concluímos a nossa jornada pelo mundo do Elasticsearch. Espero que eu tenha podido fornecer uma boa leitura e fonte de informações, que permita que você possa usar sem problemas essas ótimas ferramentas em seus projetos. Obrigado por me acompanhar neste livro, sucesso a todos e até a próxima!

Ficou com alguma dúvida? Não hesite em participar do fórum da Casa do Código, utilizando a tag "elasticsearch". http://forum.casadocodigo.com.br